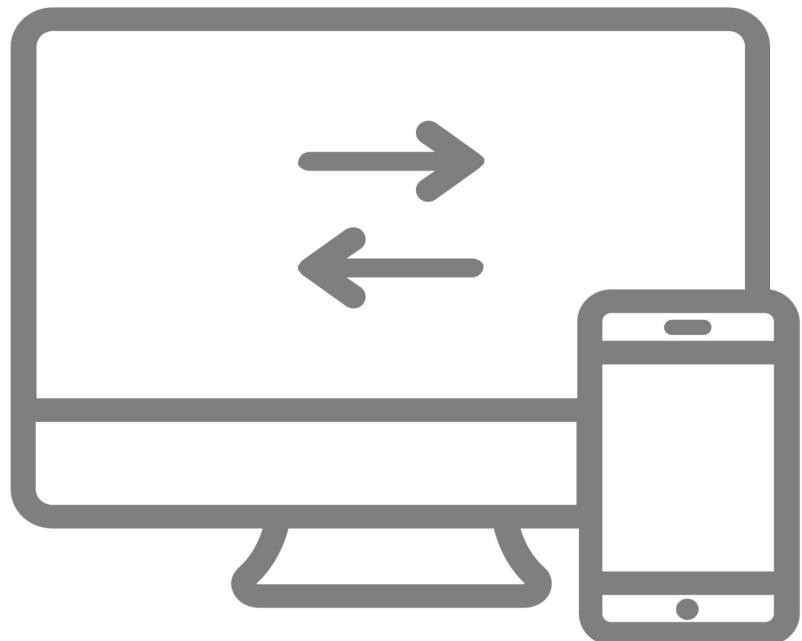


Web Apps com JavaScript Moderno, DOM e jQuery

Curso WD-47



Sumário

1 Sobre o curso	2
1.1 Os exercícios	2
1.2 O projeto	3
1.3 Tirando dúvidas	3
2 Começando um projeto front-end como um profissional	5
2.1 Escrevendo código em menos tempo	5
2.2 Developer Tools	9
2.3 Desacoplando CSS do HTML	11
2.4 Design Responsivo	13
2.5 Mobile First	16
2.6 Progressive Enhancement	16
2.7 Flexbox e o Progressive Enhancement	17
3 As funcionalidades, o JavaScript e o CSS	22
3.1 Flexbox: alterando a direção	22
3.2 Javascript, a linguagem do navegador	23
3.3 DOM: sua página no mundo JavaScript	24
3.4 Desacoplando o Javascript do CSS	27
3.5 Relembrando Eventos JavaScript	29
3.6 Javascript onde?	30
3.7 Funções anônimas	30
3.8 Ouvindo eventos em vários elementos	31
3.9 Usando a estrutura do DOM ao nosso favor	31
3.10 Desacoplando nosso código da estrutura	32
3.11 Removendo Elementos do DOM e relembrando setTimeout	34
3.12 CSS3 Transitions	35
4 jQuery	37
4.1 Conhecendo o jQuery	37

Sumário	Caelum
4.2 Eventos	40
4.3 Navegação no DOM com jQuery	42
4.4 Modificando o DOM com jQuery	42
4.5 Funções mais comuns do jQuery	43
4.6 Construindo elementos com jQuery	44
5 AJAX e a vida assíncrona	48
5.1 AJAX com jQuery	48
5.2 JSON - JavaScript Object Notation	50
5.3 \$.getJSON	50
5.4 Same origin policy e CORS	51
5.5 Same origin policy e JSONP	53
6 Melhorando nosso app com boas práticas de código	55
6.1 O problema dos escopos em JavaScript	55
6.2 IIFE: Immediately Invoked Function Expressions	56
6.3 Organização de arquivos JavaScript	56
6.4 Módulos em JavaScript	57
6.5 Módulo com objetos	59
6.6 Dependências com IIFE	61
6.7 Use strict	62
7 O Poder dos Eventos	63
7.1 Acoplamento de código	63
7.2 Eventos personalizados	64
7.3 ContentEditable	66
7.4 Elementos interativos e o foco	67
7.5 Eventos e performance do site	69
7.6 Delegação de eventos	72
8 Exercício: Removendo cartões com JavaScript	74
8.1 Objetivo	74
8.2 Passo a passo com código	75
9 Exercício: Mudando o conteúdo do botão quando ele é clicado	76
9.1 Objetivo	76
9.2 Passo a passo com código	77
10 Exercício: Mudando o layout dos cartões no mural	78
10.1 Objetivo	78
10.2 Passo a passo com código	79

11 Exercício: Funcionalidades com Progressive Enhancement	81
11.1 Objetivo	81
11.2 Passo a passo com código	81
12 Exercício: Animando a remoção do cartão	83
12.1 Objetivo	83
12.2 Passo a passo com código	83
13 Exercício: Adicionando evento de remover para todos os cartões	86
13.1 Objetivo	86
13.2 Passo a passo com código	86
14 Exercício: Mostrar opções somente do cartão que estamos mexendo	88
14.1 Objetivo	88
14.2 Passo a passo com código	88
15 Exercício: Delegando quem vai mudar a cor dos cartões.	91
15.1 Objetivo	91
15.2 Passo a passo com código	91
16 Exercício: Ajustando a navegação via teclado.	93
16.1 Objetivo	93
16.2 Passo a passo com código	93
17 Exercício: Removendo os cartões com delegate.	95
17.1 Objetivo	95
17.2 Passo a passo com código	95
18 Exercício: Validando antes de criar cartões.	97
18.1 Objetivo	97
18.2 Passo a passo com código	97
19 Exercício: Usando jQuery para criar cartões de forma Sensacional.	99
19.1 Objetivo	99
19.2 Passo a passo com código	99
20 Exercício: Devolvendo os eventos para os cartões.	102
20.1 Objetivo	102
20.2 Passo a passo com código	102
21 Exercício: Mostrando instruções para os usuários.	105
21.1 Objetivo	105
21.2 Passo a passo com código	105

22 Exercício: Alert é horrível. As instruções podem ser cartões.	107
22.1 Objetivo	107
22.2 Passo a passo com código	107
23 Exercício: O Module Pattern.	111
23.1 Objetivo	111
23.2 Passo a passo com código	112
24 Exercício: JavaScript Moderno em navegadores pré-históricos.	114
24.1 Objetivo	114
24.2 Passo a passo com código	115
25 Exercício: Trazendo as instruções com AJAX.	118
25.1 Objetivo	118
25.2 Passo a passo com código	118
26 Exercício: Salvando os cartões no servidor.	120
26.1 Objetivo	120
26.2 Passo a passo com código	121
27 Exercício: Carregando os cartões do servidor.	123
27.1 Objetivo	123
27.2 Passo a passo com código	123
28 Exercício: Compatibilidade nas APIs do JavaScript com Polyfill.	125
28.1 Objetivo	125
28.2 Passo a passo com código	126
29 Apêndice - Dando poderes ao conteúdo	127
29.1 Transformando textos em outros textos	127
29.2 String.replace()	127
29.3 Expressão Regular em JavaScript	127
29.4 Exercício: Cartões mais poderosos com Expressões Regulares	129
29.5 Medidas relativas: em	129
29.6 Alinhamento com flexbox	131
29.7 O forEach do ES5	132
29.8 Exercício: Melhorando a vizualização dos cartões	133
29.9 Mais funções do jQuery	135
29.10 Expressões regulares dinâmicas	135
29.11 Exercício: Buscando cartões com jQuery	136
30 Apêndice - Automatização de Tarefas	137

30.1 Um pouco sobre Node.js	137
30.2 Instalando Gulp	139
30.3 Gulpfile e Tasks	140
30.4 Exercício: Instalando Gulp e a primeira task	140
30.5 Prefixos automáticos	141
30.6 Copiando arquivos	142
30.7 Exercício: Copy e o Autoprefixer	143
30.8 Dependências em tasks e a task Default	144
30.9 Exercício: Melhorando nossas tasks	144
30.10 Gulp watch	145
30.11 Exercício: Automatizando a automatização com watch	146
31 Apêndice - Descomplicando o CSS com SASS	147
31.1 Pré-processadores CSS	147
31.2 SASS	147
31.3 Compilando SASS com Gulp	148
31.4 Exercícios: Nesting e SASS com Gulp	149
31.5 Reaproveitamento com mixins	150
31.6 Exercícios: Isolando código em mixins	151
31.7 Discussão em aula: Preciso mesmo de um pré-processador? Quando o CSS é o suficiente?	152

Versão: 21.5.7

SOBRE O CURSO

Com a evolução das tecnologias mobile, criou-se novas formas de se acessar informações e serviços web. Hoje, para manter competitividade de mercado, as aplicações web devem se adaptar não apenas a vários tamanhos de tela, mas às diferentes formas que o usuário pode interagir com a aplicação.

Há alguns anos, o JavaScript desempenhava um papel secundário entre as ferramentas utilizadas pelos profissionais de desenvolvimento de aplicações Web, apesar de ter sido criado especificamente para melhorar a interação do usuário com as aplicações, através da possibilidade de adicionarmos código que roda diretamente no cliente, ou seja, no navegador.

Nos últimos anos, porém, o JavaScript tem ganhado cada vez mais espaço e importância nesse papel, e inclusive vem sido usado no lado do servidor de aplicações e já não é tratado como um simples coadjuvante. Dentre os casos de sucesso do uso do JavaScript, podemos destacar as aplicações de produtividade do Google: o GMail, Google Calendar e a suíte Google Docs. Além de utilizarem JavaScript no lado do servidor, o uso da linguagem no lado do cliente garante que tenhamos as facilidades de uso e interações avançadas, partes estratégicas no sucesso desses produtos.

Esse cenário só vem mudando graças às evoluções das tecnologias e ferramentas de desenvolvimento, à evolução dos navegadores, à padronização da plataforma Web como um todo e também às evoluções da linguagem em si.

Esse curso pretende apresentar ao aluno o mundo do desenvolvimento JavaScript no front-end, suas técnicas e ferramentas, para que ele possa compreender como podem ser melhor utilizadas e ter suas funcionalidades melhor exploradas.

1.1 OS EXERCÍCIOS

Os exercícios são essenciais para a melhor compreensão dos tópicos apresentados, pois neles vamos aplicar as técnicas recomendadas da linguagem e vamos analisar e solucionar, de maneira prática, os problemas que enfrentamos no dia a dia como desenvolvedores. Além dos exercícios que fazem parte da proposta didática, apontaremos no decorrer do curso alguns exercícios opcionais, sugerimos que os mesmos sejam feitos pois neles iremos explorar algumas alternativas de solução de problemas para conhecermos melhor algumas características do JavaScript e das bibliotecas que fazem parte do curso.

Além dos exercícios também apontaremos no decorrer do curso alguns desafios para que você

coloque em prática a sua capacidade analítica na solução dos problemas. Alguns desafios irão requerer alguma pesquisa e leitura de documentação, então fique atento às referências apresentadas na apostila e pelo instrutor durante o curso.

1.2 O PROJETO

Durante todo o curso, trabalharemos na criação de uma aplicação que funcionará como um mural de notas/cartões. Faremos apenas uma página, e ela terá toda a dinamicidade necessária para o usuário utilizar sem precisar recarregar o html no navegador. Além disso, nossa aplicação será implementadas levando em conta algumas restrições do usuário em diferentes plataformas.

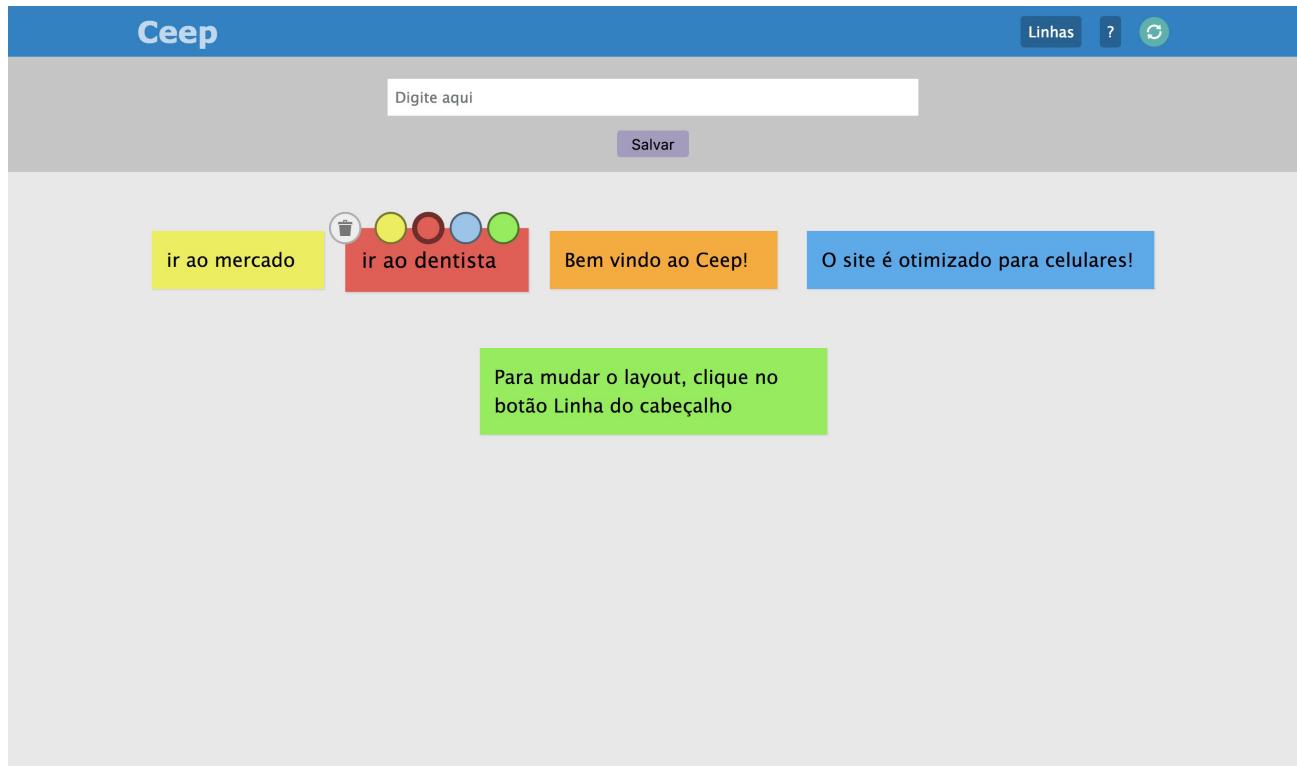


Figura 1.1: Projeto visto num desktop

1.3 TIRANDO DÚVIDAS

Durante o curso, tenha a certeza de tirar todas as suas dúvidas com o instrutor. Algumas características do JavaScript podem parecer básicas a princípio, mas conhecê-las a fundo é essencial para seu desenvolvimento e compreensão de alguns tópicos mais avançados. Não tenha medo de fazer perguntas, por mais básicas que elas possam parecer, são esses pontos básicos os mais importantes para a melhor compreensão dos assuntos abordados.

Além disso, recomendamos fortemente a busca de recursos externos para seu aprendizado, como por exemplo livros, websites e blogs, a participação em fóruns e listas de discussão relacionadas ao assunto.

Por exemplo o GUJ que, apesar de ter nascido no mundo do desenvolvimento em Java, hoje conta com a participação de milhares de profissionais das mais variadas áreas, inclusive JavaScript. O portal iMasters é outro ponto de referência no assunto.

A seguir apresentamos algumas referências importantes:

- GUJ (<http://www.guj.com.br>)
- iMasters (<http://www.imasters.com.br/secao/javascript>)
- Grupo de usuários JavaScript Brasil (<http://groups.google.com/group/javascript-bra>)
- Grupo de usuários Node.js (<http://groups.google.com/group/nodebr>)
- Livro - Pro JavaScript Techniques (RESIG, John) *em inglês somente*
- Livro - O Melhor do JavaScript (CROCKFORD, Douglas)
- Livro - Secrets of the JavaScript Ninja (RESIG, John; BIBEAULT, Bear)

CAPÍTULO 2

COMEÇANDO UM PROJETO FRONT-END COMO UM PROFISSIONAL

2.1 ESCREVENDO CÓDIGO EM MENOS TEMPO

Estamos prestes a começar nossa app. A página principal, onde serão exibidos os cartões, é nosso primeiro objetivo.

Toda página html tem certos elementos em comum, como o `DOCTYPE`, a tag `meta` para a configuração de encoding e as tags `head` e `body`. A nossa não será diferente, e vai ficar com essa cara:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Título da página</title>
</head>
<body>
  Conteúdo da página aqui.
</body>
</html>
```

Esse é um código importante e que é repetido em todo início de projeto. Será que todo mundo digita isso na mão, toda hora?

Como profissionais, estamos sempre buscando uma forma de melhorar nossa produtividade. Atividades comuns e repetitivas como criação de tags e esqueletos de páginas não podem ocupar muito tempo de desenvolvimento.

Ao longo do curso criaremos e editaremos **muito** código. Usar um bloco de notas comum para iniciar o projeto do curso acrescentaria muito tempo somente à digitação do código.

Editores de código e plugins

Com o passar do tempo e o costume com as tecnologias, a linguagem deixa de ser a maior barreira para implementação das funcionalidades de um sistema. Em um dado momento, é comum que a velocidade de pensamento ultrapasse nossa capacidade de produzir código.

Produzir código é digitar, digitar é lidar com o editor de texto. Com o tempo, muitos editores de texto evoluíram tentando otimizar o tempo que passamos digitando. Hoje em dia temos muitos editores

para edição de código. Aqui, uma lista com alguns deles:

- Sublime
- Atom
- Brackets
- Notepad++
- Visual Studio Code

Qual escolher? Todos os editores acima são muito bons para edição de código. Mas qual tipo de código?

Temos muitas linguagens e diversas formas de desenvolvimento. Se esses editores resolvessem todos os problemas, de todas as linguagens, de uma vez, eles seriam enormes e pesados.

Para adaptação a cada linguagem e padrão de desenvolvimento, esses editores optaram por distribuir funcionalidades muito específicas em **plugins**.

A vantagem da utilização de plugins é que eles não pertencem a nenhum editor específico. São programas independentes, que adicionam funcionalidades ao editor.

Emmet

Para digitar html e css de forma mais rápida, vamos usar uma fantástica ferramenta chamada Emmet. O Emmet é um plugin que consegue gerar códigos, precisando apenas de alguma dica nossa para saber o que escrever.

Para escrever o código do esqueleto da página usando o Emmet basta digitar `!` e em seguida pressionar **TAB**. O **TAB** fala pro Emmet completar nosso código. Como já escrevemos `!`, ele sabe que queremos um `<!DOCTYPE>`, ou seja, queremos uma nova página. Ele gera toda a estrutura básica, incluindo `head`, `body`, `title` etc.

Outros atalhos do Emmet

Podemos usar o Emmet para gerar mais que só o esqueleto do nosso HTML. Por exemplo, precisamos criar uma lista não ordenada com 6 itens. Normalmente começamos escrevendo a tag ``, seguido das tags ``, não esquecendo de fechar cada uma delas:

```
<ul>
<li>Item 1</li>
<li>Item 2</li>
<li>Item 3</li>
<li>Item 4</li>
<li>Item 5</li>
<li>Item 6</li>
</ul>
```

Podemos fazer isso muito mais facilmente com o Emmet. Para criar uma tag, basta escrever o nome

dela e apertar **TAB**. Assim o código:

```
ul
```

gera o código:

```
<ul></ul>
```

Para criar os `` podemos fazer a mesma coisa. Mas precisamos de 6 deles. Para fazer de forma rápida, podemos colocar um multiplicador para o Emmet saber quantos elementos criar:

```
<ul>
  li*6
</ul>
```

E apertamos **TAB**:

```
<ul>
  <li></li>
  <li></li>
  <li></li>
  <li></li>
  <li></li>
  <li></li>
</ul>
```

Note que desde o começo, queríamos colocar os `li`s dentro da `ul`. Também podemos fazer isso de uma vez com o Emmet ao invés de fazer por partes, usando o símbolo `>`:

```
ul>li*6
```

TAB de novo:

```
<ul>
  <li></li>
  <li></li>
  <li></li>
  <li></li>
  <li></li>
  <li></li>
</ul>
```

Muito bom, não?

Note que precisamos ter o cursor no final da expressão, pois o Emmet ignora tudo o que está depois dele. Por exemplo, se usarmos a mesma expressão acima, mas colocar o cursor entre o `li` e o `*6` ele vai gerar algo assim:

```
<ul>
  <li></li>
</ul>*6
```

Falta colocar o conteúdo nos `li`s. Nossa conteúdo é muito parecido. Podemos mandar o Emmet adicionar conteúdo para gente usando `{conteúdo}` logo depois do nome da tag, por exemplo:

```
li{Item 1}
```

E ao apertar TAB:

```
<li>Item 1</li>
```

Se o conteúdo for igual para todos os `li`s, podemos ainda mesclar as instruções de conteúdo com o multiplicador:

```
li{Item 1}*6
```

vira:

```
<li>Item 1</li>
<li>Item 1</li>
<li>Item 1</li>
<li>Item 1</li>
<li>Item 1</li>
<li>Item 1</li>
```

Mas não queremos o número do item igual, queremos item de 1 a 6. Para isso o Emmet tem o símbolo `$` que serve como um contador que começa em 1 e vai incrementando:

```
li{Item $}*6
```

vira:

```
<li>Item 1</li>
<li>Item 2</li>
<li>Item 3</li>
<li>Item 4</li>
<li>Item 5</li>
<li>Item 6</li>
```

E o Emmet é ainda mais poderoso. Queremos agora dar o id "lista" para o `ul` e a classe "item" para cada `li`. Podemos dar atributos para os elementos que o Emmet vai criar usando os mesmos **seletores do CSS**. Ou seja, para criar a `ul` já com id, usamos a expressão `ul#lista`. E para criar os `li`s com classe, fazemos `li.item`.

Usando todas essas funcionalidades juntas, podemos rapidamente escrever um código longo e repetitivo, e sem o risco de cometer erros bobos:

```
ul#lista>li.item{Item $}*6
```

Apertar TAB:

```
<ul id="lista">
  <li class="item">Item 1</li>
  <li class="item">Item 2</li>
  <li class="item">Item 3</li>
  <li class="item">Item 4</li>
  <li class="item">Item 5</li>
  <li class="item">Item 6</li>
</ul>
```

Instalando plugins em casa

Cada editor tem o seu jeito de instalar plugins. Vamos abordar aqui apenas alguns dos principais.

Sublime

Para facilitar a instalação de plugins, instale um plugin chamado *Package Control* seguindo as instruções desse site: <https://packagecontrol.io/installation>.

Com o *Package Control* instalado, utilizaremos o atalho para acessar qualquer funcionalidade do Sublime, o **ctrl+shift+P**. No campo que abre, procure e selecione a opção *install package*. Ele vai abrir um novo campo com a mesma cara. Agora basta procurar o plugin pelo nome e selecionar a opção desejada para instalar.

Atom

Selecione a opção *Edit > Preferences* no menu. Na nova janela, escolha no menu a esquerda a opção "+ install". O Atom vai abrir um campo de busca. Nesse campo você pode procurar o plugin pelo nome e instalá-lo.

Brackets

Selecione a opção *Extension Managements* no menu *File*. Novamente, basta procurar o plugin pelo nome e clicar em *install*.

2.2 DEVELOPER TOOLS

Agora que temos um esqueleto do HTML, podemos começar a popular nosso <body> com conteúdo.

Faremos o código de que primeiro? Não existe ordem correta para isso. No curso, implementaremos cada funcionalidade do app completamente antes de partir para a próxima funcionalidade. E vamos começar com o cabeçalho.

Precisamos seguir o layout que o designer criou. Para o fundo, ele pediu uma cor bastante específica, a mesma do site da Caelum. Mas como pegar essa cor? Ora, como todo site, o site da Caelum também é feito com HTML e CSS, portanto precisamos de um jeito de acessar essas informações.

Hoje em dia, todos os navegadores já possuem alguma ferramenta que nos permite acessar diversas informações a respeito do site que está aberto nele, entre elas o HTML fonte. Esse tipo de ferramenta chama-se **Developer Tools** (ou DevTools).

A forma mais fácil de pegar a nossa cor é abrir o site da Caelum, clicar com o botão direito na cor que queremos e escolher a opção "*Inspect Element*". O navegador irá abrir o Developer Tools dele, mostrando o HTML fonte com o foco no elemento que mandamos inspecionar.

Aulas com exclusiva e reconhecida didática da Caelum. Aprenda as tecnologias que fazem diferença no mercado. Instrutores altamente capacitados e com destaque na comunidade.

Confiamos tanto em nossos cursos que 90% dos instrutores são ex-alunos da Caelum!

Conheça os cursos da Caelum

Java Mobile Front-end Agile .NET

OO, Web, JSP, JSF, Java EE Android, iOS, Games, Web responsiva HTML, CSS, JavaScript Scrum, práticas ágeis C#, ASP.NET MVC, NHibernate, Unity PHP e MySQL

Jornada Back-end Um programa completo para uma carreira excepcional em programação 6 cursos Java + 1 TCC prático + 1 Diploma diferenciado Conheça

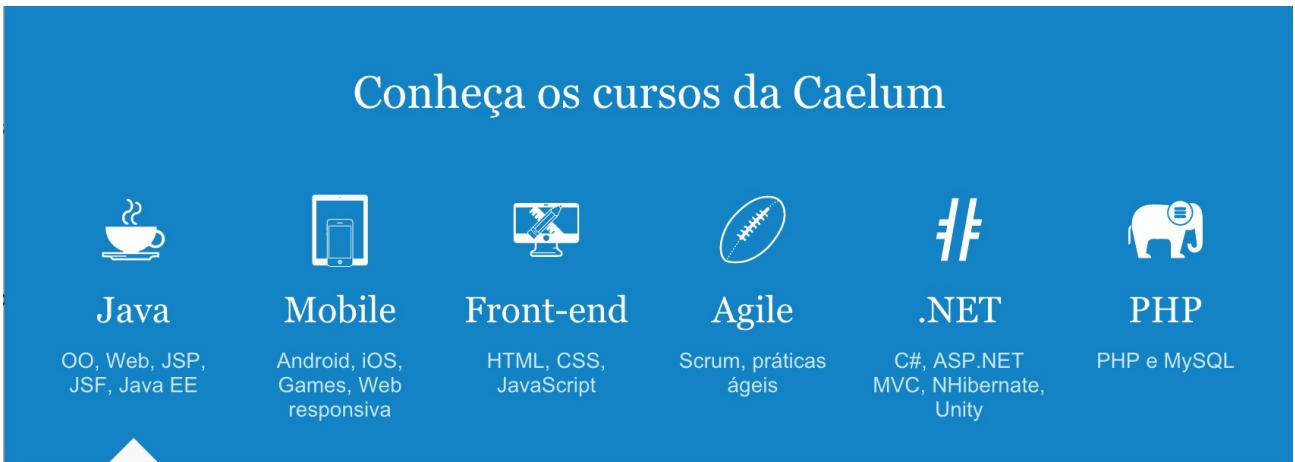
Back Forward Reload Save As... Print... Translate to English View Page Source View Page Info Inspect Element

O que os alunos falam

Quem já treinou aqui

Mas ver o HTML fonte não é o bastante. Se o site foi feito seguindo as boas práticas, a cor que queremos não vai está lá, e sim num arquivo CSS. Sabemos que CSS funciona com seletores. E agora? Qual seletor será que o pessoal do site da Caelum usou para atribuir a cor ao elemento que selecionamos? Tag, id, class... tem muitas possibilidades!

Áí entra de novo o Developer Tools. Para renderizar a página, o navegador teve que ler todos o CSS e aplicar os estilos nos respectivos elementos. Por isso, ao inspecionar um elemento, ele já nos mostra também quais estilos estão aplicados nele. Ele também nos mostra em que arquivo está o estilo e qual o seletor usado para aplicá-lo ao elemento.



The screenshot shows the Chrome DevTools Elements tab. On the left, the DOM tree highlights the element `<div class="home-cursos">`. On the right, the Styles panel displays the CSS rules applied to this element, including:

```

Styles Computed Event Listeners »
element.style {
}
@media (min-width: 43em)
  .home-cursos {
    padding: 2.5em 0;
}
.home-cursos {
  background: #0082c7;
  padding: 1em 0;
  text-align: center;
}
*, :after, :before {
}

```

The `background: #0082c7;` rule is highlighted in blue, indicating it is currently selected. A search bar at the bottom right of the panel says "Find in Styles".

Além disso, ao focar em um elemento no HTML, o navegador o destaca na página, mostrando também os espaçamentos relacionados a ele (margin, border, padding e conteúdo).

O DevTools é uma ferramenta muito poderosa para o desenvolvedor front-end. Ela não só nos permite identificar rapidamente informações sobre os elementos da página, como ainda permite alterações e ver em tempo real como elas afetam o resultado final.

2.3 DESACOPLANDO CSS DO HTML

Agora que já temos a cor que queremos no cabeçalho, podemos criá-lo. Nosso cabeçalho é bem simples, consistindo apenas no nome da aplicação e a cor de fundo. Portanto, podemos muito bem fazer esse código usando um simples `h1`:

```
<h1>Ceep</h1>
```

E o seu respectivo estilo:

```
h1 {
  background: #0082c7;
}
```

Pronto!

Mas será que `h1` é a melhor escolha? É muito comum que aplicações tenham mais informações no

cabeçalho que apenas o nome. Menu de navegação ou menu de opções são os conteúdos mais frequentes.

Se formos colocar um menu desses no nosso cabeçalho, é melhor que ele seja uma tag mais representativa, como a `header`, e fazer ele conter o nome e o menu:

```
<header>  
  <h1>Ceep</h1>  
  
  <nav>  
    ...  
  </nav>  
  
<header>
```

Mas agora, quem está com o fundo na cor que queríamos para o cabeçalho? Apenas o `h1`! Temos que mudar também no **estilos.css**:

```
header {  
  background: #0082c7;  
}
```

Peraí, a escolha da tag que vamos usar para o cabeçalho faz parte do conteúdo da nossa aplicação. Uma mudança de conteúdo deveria causar uma mudança no CSS? Vamos lembrar que a boa prática é deixar CSS só com o estilo. O ideal é deixá-lo independente das tags HTML.

A melhor forma de fazer isso é com classes. Onde queremos essa cor na nossa aplicação? No **cabeçalho**! Então podemos criar uma classe que vai representar o cabeçalho, e aplicar a cor nele:

```
.cabecalho {  
  background: #0082c7;  
}
```

Agora, basta dar essa classe para o elemento HTML que vai ser o nosso cabeçalho. O **h1** no primeiro caso e **header** no segundo:

```
<h1 class="cabecalho">Ceep</h1>
```

ou

```
<header class="cabecalho">  
  <h1>Ceep</h1>  
  
  <nav>  
    ...  
  </nav>  
  
<header>
```

Dessa forma, sempre teremos a aplicação com o mesmo estilo, **não importa qual elemento HTML estamos usando para representá-lo**. Nosso estilo e nosso conteúdo estão *desacoplados*.

2.4 DESIGN RESPONSIVO

Ao se falar em aplicações web, ainda é natural imaginar usuários acessando através de um computador ou um notebook. Mas na verdade, em vários cenários, usamos mais dispositivos mobile do que desktop. Por isso, ao desenvolver é importante que se tenha em mente a usabilidade nos diversos tipos de ambiente que possamos encontrar.

Um site que se adapta às diversas formas e limitações dos dispositivos é um site **Responsivo**.

Media Query

É bem difícil que os estilos de uma página permaneçam iguais entre dispositivos diferentes. Afinal, cada dispositivo tem suas características e limitações. Nossa site deve se adaptar a elas.

Num celular, temos restrições com a largura da tela. Nesses casos, é muito comum que elementos sejam empilhados, um abaixo do outro, aproveitando melhor o espaço vertical da tela. Esse tipo de alinhamento é bem diferente do alinhamento que utilizamos numa tela maior, onde elementos também são alinhados um do lado do outro, aproveitando o espaço horizontal.

Dado que estamos implementando um design responsivo, nossa aplicação deve exibir o mesmo conteúdo com esses alinhamentos diferentes em cada caso (tamanhos de tela). Precisamos de estilos exclusivos para cada caso. Para isso, existem as **media queries**.

Usando Media Queries, podemos criar CSS que só será aplicado em alguns casos. Por exemplo, queremos que em telas grandes as seções de uma página se organizem em 2 colunas.

```
.secao {  
    display: block;  
}  
  
@media (min-width: 800px){  
  
    .secao {  
        display: inline-block;  
        width: 45%;  
    }  
  
}
```

Aqui, em telas maiores que 800px, a classe `secao` vai ganhar as duas propriedades que farão a configuração de duas colunas. Nas telas abaixo disso, esse CSS é ignorado e vai seguir o estilo padrão, de um elemento em cima do outro.

Viewport

Chamamos de viewport o espaço disponível para o site ser renderizado no browser.

Media queries que usam medidas de largura ou altura estão sempre se referindo às dimensões do

viewport.

```
@media (min-width: 650px){  
    .baleia {  
        display: inline-block;  
    }  
}
```

O estilo acima será aplicado quando o viewport tiver mais do que 650 pixels de largura. Como saber a largura em pixels do viewport do aparelho? Podemos olhar para a resolução da tela.

Um iPhone 6 tem resolução de 1334 x 750 em uma tela de 4.7", resolução maior que muitos notebooks com telas maiores. Isso só é possível porque os pixels físicos do iPhone são muito menores que os pixels físicos de um notebook.

Se o site for renderizado com base na resolução da tela, 1px no css seria equivalente a 1 pixel físico da tela. Teríamos um site com tudo muito pequeno. Por isso, não usamos a resolução do celular, mas sim, uma medida que o próprio aparelho define como sendo seu tamanho de viewport ideal, uma dimensão em pixels menor que a resolução.

Para alterar a medida do nosso viewport criaremos a seguinte tag `<meta>` :

```
<meta name="viewport" content="width=device-width">
```

Medidas flexíveis: % e viewport units

Em telas pequenas, elementos com dimensões em pixels podem não caber na tela. Nesses casos, uma alternativa aos pixels são as medidas flexíveis do CSS.

Uma unidade muito utilizada é a %, que já usamos no exemplo de media query acima. Ela representa uma parcela (em porcentagem) da altura ou largura do contexto no qual o elemento está inserido, que geralmente é o elemento pai.

```
<body>  
    <div class="contexto">  
        <div class="elemento">  
            Oi  
        </div>  
    </div>  
</body>  
  
.contexto {  
    width: 90%;  
}  
.elemento {  
    width: 50%;  
}
```

No código acima, a `<div class="contexto">` é sempre 10% menor que a largura do `<body>`. A `<div class="elemento">` está sempre com metade do tamanho da `<div class="contexto">`.

Há casos nos quais é necessário que nosso elemento fique exatamente com a altura ou largura da página. Trabalhar com porcentagem nesses casos implica que o elemento seja filho direto da tag `<body>`, o que pode impactar na semântica do site. Para esses casos, pode-se utilizar algumas unidades muito parecidas com a porcentagem, mas que, independente do lugar no qual o elemento está inserido, são sempre relativas ao tamanho do viewport. São as *Viewport Units*:

- `vw` - % da largura do viewport
- `vh` - % da altura do viewport
- `vmin` - % do menor entre largura e altura
- `vmax` - % do maior entre largura e altura

TIPOGRAFIA RESPONSIVA

O que acontece se colocarmos o `font-size` em porcentagem, como no exemplo abaixo?

```
.elemento {  
    font-size: 150%;  
}
```

O tamanho da fonte será 150% da largura ou da altura? Nenhum dos dois, no caso acima, a fonte será 1.5 vezes maior que o `font-size` do elemento pai. Ou seja, o tamanho da fonte não está sendo alterado de acordo com o tamanho do viewport.

Para que isso aconteça, podemos usar unidades relativas ao viewport.

```
.elemento {  
    font-size: 1.5vw;  
}
```

O `font-size` do `.elemento` é 1.5% da largura do viewport.

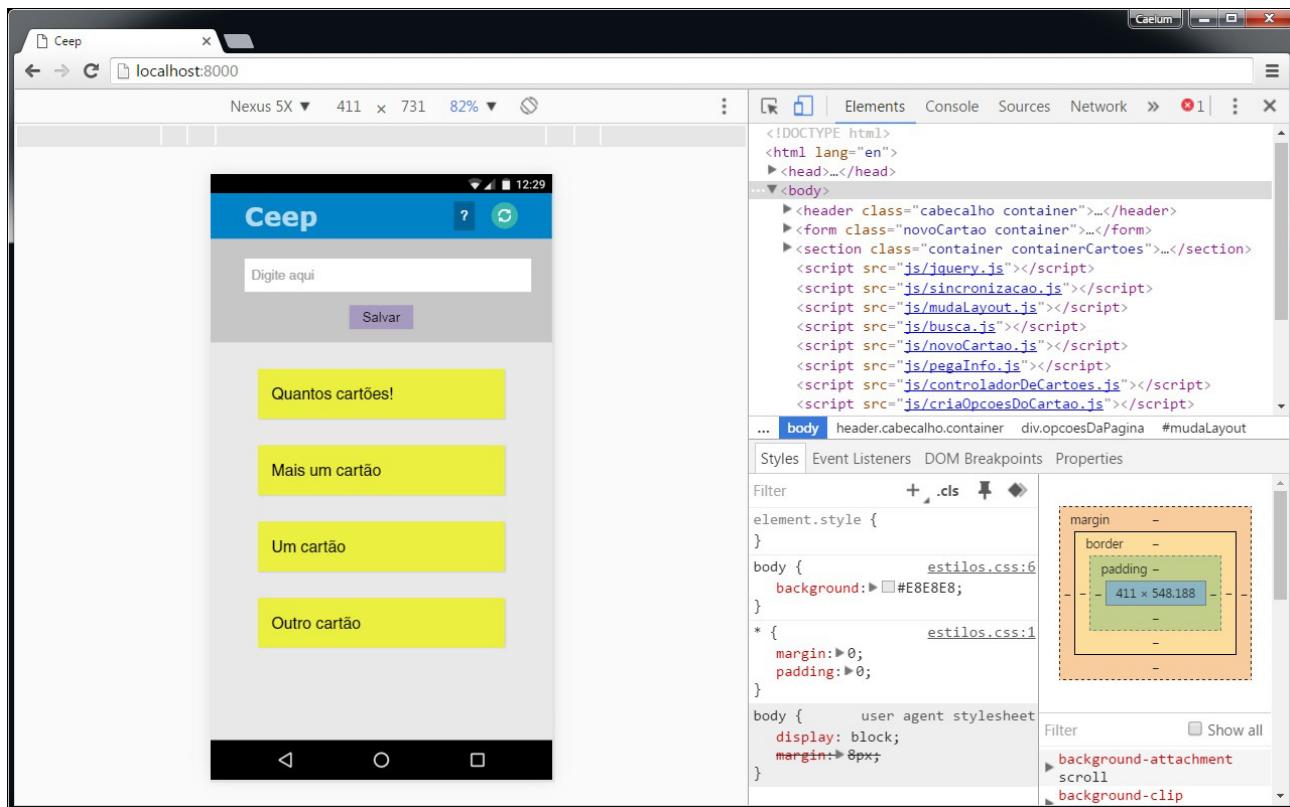
Responsive Mode e Device Mode

Como fazer para testar o funcionamento da nossa página em diversos aparelhos? Podemos abrir nosso site em algum celular e testar. Porém, testar em diferentes tamanhos de tela acaba sendo bem trabalhoso.

Outra solução é testar o site diminuindo o tamanho da janela do navegador. Porém, em diversas situações, precisamos testar o site em um viewport pequeno, com a ajuda do Developer Tools. Como se virar num espaço tão pequeno?

Pensando nisso, os navegadores criaram uma ferramenta dentro do Developer Tools chamada "Responsive Mode", que nos permite configurar o tamanho do viewport que queremos testar, ao mesmo tempo que mantém o tamanho original da janela. Existem até alguns tamanhos pré-programados

simulando os dispositivos mobile mais comuns do mercado.



No Chrome o Responsive Mode evoluiu para fazer muito mais do que apenas simular um viewport pequeno e passou a se chamar Device Mode.

É possível emular diversas outras características de hardware que variam de device para device, como tela sensível ao toque, conexão ruim, dados dos sensores de geolocalização e acelerômetro.

2.5 MOBILE FIRST

No exercício anterior, deixamos o layout pronto para um grupo de usuários mais limitados, os usuários que tem telas pequenas. Estamos seguindo a técnica chamada Mobile-First.

Um site que cabe numa tela grande não necessariamente cabe na tela de um celular. Já um site que consegue disponibilizar seu conteúdo principal e funcionalidades básicas em telas pequenas, com certeza, já consegue numa tela grande.

Esse é o grande motivo de começarmos nosso projeto pensando no mobile primeiramente. Garantir que o conteúdo esteja disponível para nossos usuários com limitação no tamanho de tela.

2.6 PROGRESSIVE ENHANCEMENT

Ao longo do curso, veremos que as telas não são o único limitador quando falamos de sites e web apps. Assim como a limitação de tela, limitações de conexão com a internet, limitações de versão de navegadores e outras, terão papel importante nas decisões de implementação do projeto, desde o início.

Desenvolver garantindo que nossas funcionalidades básicas e conteúdo estejam disponíveis, não necessariamente da mesma forma, para nossos usuários mais limitados, desde o começo, é a definição de Progressive Enhancement.

2.7 FLEXBOX E O PROGRESSIVE ENHANCEMENT

Não existe nenhuma tag específica que possa representar os cartões que vamos usar na aplicação. Então decidimos por usar uma `div` para cada, com uma classe `cartao` para trabalharmos no estilo deles.

```
<div class="cartao">
```



Quando criamos vários cartões, cada um fica em uma linha, pois o `display` padrão da `div` é `block`. Se queremos mais de um cartão por linha, podemos simplesmente aplicar o valor `inline-block`:

```
.cartao {  
  display: inline-block;  
}
```

Mas peraí! Mesmo assim os cartões ainda estão ocupando a linha toda. Isso porque a largura dos elementos não está definida, então naturalmente cada um ocupa todo o espaço disponível. Para caber mais de um na mesma linha, temos que definir uma largura fixa:

```
.cartao{  
  display: inline-block;  
  width: 190px;  
}
```



Agora cabem vários cartões na mesma linha. E olhe só, automaticamente o navegador já calcula quantos cartões cabem e adapta de acordo. Nossa aplicação está responsiva e não precisamos fazer nada a mais.

Bom, mais ou menos. Quando usamos o `inline-block` o navegador começa posicionando os elementos na ordem de leitura, no nosso caso, da esquerda pra direita e quando não cabe mais, o próximo elemento vai embaixo. Assim nossos cartões acabam descentralizados. Como resolver esse problema?

Queremos o tamanho dos cartões fixo, então temos que colocar margens para alinhá-los. Mas quanto de margem? Conseguimos calcular pra ficar certinho no centro? Não sabemos nem a largura do container! E quando o número de cartões na linha mudar? Teremos que usar uma combinação de margens relativas e media queries só pra garantir que tudo fique centralizado.

O problema que estamos tendo aqui é que queremos **todos os cartões** centralizados, ou seja, queremos um comportamento que depende um do outro, um comportamento **em grupo**. Para isso entra o **Flexbox**. O flexbox é um tipo de display pensado para dizer como os filhos de um elemento devem se posicionar dentro dele, como um conjunto, e ele é responsável em fazer os cálculos necessários.

Então, para usarmos o flexbox para posicionar os cartões, precisamos que eles estejam dentro de um mesmo elemento. Usamos, então o `<section>`, para separar essa informação do resto da aplicação.

```
<section class="mural">
  <div class="cartao">...</div>
  <div class="cartao">...</div>
  <div class="cartao">...</div>
  <div class="cartao">...</div>
  <div class="cartao">...</div>
</section>
```

Agora, vamos falar para o `mural` como os cartões devem se posicionar dentro dele, usando o flexbox:

```
.mural{
```

```
        display: flex;
    }
```



Ok, não era bem isso que queríamos. A função do flexbox é **fazer os elementos caberem** no espaço separado para eles. Nesse caso, ele redimensionou todo mundo pra caber em uma única linha. Agora precisamos falar pra ele que ele deve manter o tamanho original e passar os elementos que não couberem para a linha de baixo. Fazemos isso com a propriedade `flex-wrap` :

```
.mural{
    display: flex;
    flex-wrap: wrap;
}
```



Para deixar nossa aplicação ainda mais responsiva, vamos dizer que cada cartão, apesar de ter tamanho padrão 190px, pode ser redimensionado para ficar maior se tiver espaço sobrando na tela, mas não couber um cartão novo. Essa é uma propriedade de cada cartão, é possível fazer alguns poderem redimensionar, enquanto outros não. No nosso caso, vamos colocar em todos:

```
.cartao {
    flex-grow: 1;
```

}



O resultado é bem impressionante! O flexbox é uma ferramenta muito versátil e simples para trabalhar com posicionamento que foi adicionada na versão CSS3.

Mas sendo uma ferramenta nova, o que acontece se o usuário estiver num navegador sem suporte a ela? Quando o navegador ler o CSS ele vai considerar que a funcionalidade não existe e vai simplesmente ignorar, assim o display vai ser renderizado usando o `inline-block` que colocamos anteriormente!

O flexbox **sobrescreve** o comportamento do `display`, mas se não existir flexbox, o `inline-block` é usado no lugar. Dessa forma, conseguimos fazer uma aplicação que funciona para ambos os usuários, independente do suporte que o navegador dele dá.

Note que começamos a fazer a aplicação mais simples, que funciona em todos os lugares **e depois adicionamos funcionalidades** apenas para os usuários que tem suporte a elas. Estamos trabalhando com **Progressive Enhancement**, ou melhoria progressiva. Quando uma funcionalidade não funciona para algum usuário, o navegador então usa uma outra funcionalidade que a substitua para manter tudo funcionando. Essa funcionalidade "reserva" recebe o nome de **fallback**.

DESCOBRINDO COMPATIBILIDADE

Sempre que vamos usar uma nova ferramenta ou propriedade, precisamos nos preocupar em como ela vai se desempenhar em todos os devices e navegadores. O "Can I Use" (<http://www.caniuse.com>) é um site mantido pela comunidade com as informações sobre o suporte oferecido às funcionalidades.

Box-sizing

O box model padrão do CSS pode ser esquisito. Por isso a propriedade `box-sizing` do CSS3 nos permite trocar o box model que queremos usar.

Por padrão, todos os elementos têm o valor `box-sizing: content-box` o que indica que o tamanho dele é definido pelo seu conteúdo apenas. Mas podemos trocar por `box-sizing: border-box` que indica que o tamanho agora levará em conta até a borda – ou seja, o width será a soma do conteúdo com a borda e o padding.

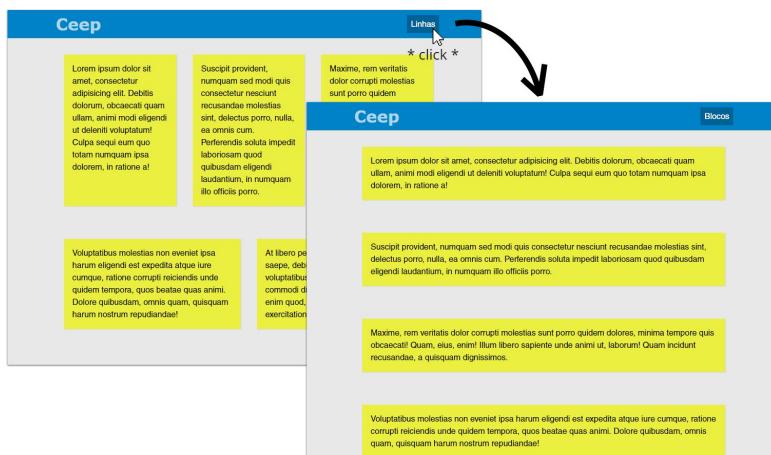
CAPÍTULO 3

AS FUNCIONALIDADES, O JAVASCRIPT E O CSS

Neste capítulo, vamos expandir as funcionalidades da nossa aplicação Ceep.

Opção para mudar o layout

Nossa aplicação terá dois modos de visualização dos cartões: um do lado do outro, como temos hoje, e outro modo de um cartão em cima do outro. Para alterar o modo de visualização, o usuário precisa clicar em um botão no cabeçalho.



3.1 FLEXBOX: ALTERANDO A DIREÇÃO

Para fazer os cartões se alinharem em linha, escorregendo quando não couber, fizemos:

```
.mural {  
  display: flex;  
  flex-wrap: wrap;  
}  
.cartao {  
  flex-basis: 190px;  
  flex-grow: 1;  
}
```

Isto é, o `mural` é declarado como um *container flex* que permite elementos escorregarem nas linhas debaixo (`flex-wrap: wrap`). Cada `cartao` possui um tamanho de 190px (`flex-basis`) mas que pode ser esticado para ocupar todo o espaço do pai (`flex-grow`).

Para mudarmos o layout para 1 cartão por linha, a essência é alterar a direção do flex. Por padrão, todo elemento é `flex-direction: row`, por isso os cartões são dispostos em linha. Podemos alterar a direção do mural da seguinte maneira:

```
.mural {  
    flex-direction: column;  
}
```

Mudar a direção do flex altera o significado da propriedade `flex-basis` que tínhamos colocado nos cartões.

```
.mural {  
    flex-direction: column;  
}  
  
.cartao {  
    flex-basis: 190px;  
    flex-grow: 1;  
}
```

O `flex-basis` refere-se sempre à dimensão que está no mesmo sentido do `flex-direction`. Ou seja, no código acima, o `flex-basis` refere-se à altura do cartão.

Para que o cartão tenha a altura ajustada dependendo do tamanho do seu conteúdo e não de um número fixo de pixels, podemos alterá-lo da seguinte forma:

```
.cartao {  
    flex-basis: auto;  
    width: auto;  
}
```

Para que a funcionalidade seja aplicada em browsers que não suportam o flexbox, podemos alterar o `display: inline-block` dos cartões para `block`.

```
.cartao {  
    flex-basis: auto;  
    width: auto;  
    display: block;  
}
```

Mas, claro, não queremos mudar isso em todos os cartões. Vamos fazer com que essas regras sejam aplicadas apenas quando o usuário clicar no botão de alteração do modo de visualização.

3.2 JAVASCRIPT, A LINGUAGEM DO NAVEGADOR

Apenas com CSS é possível ver que um elemento foi clicado e alterar o estilo de outros elementos em resposta:

```
<button id="botaoMudaLayout">Muda Layout</button>  
<section class="mural">  
    <div class="cartao"></div>  
    <div class="cartao"></div>  
</section>
```

```

#botaoMudaLayout:active + .mural {
    flex-direction: column;
}

#botaoMudaLayout:active + .mural .cartao {
    flex-basis: auto;
    width: auto;
    display: block;
}

```

Note que esses estilos dependem totalmente do posicionamento do botão no html. Nossa botão será colocado dentro de um `<header class="cabecalho">`, assim, esses estilos já não funcionarão mais.

```

<!-- O seletor #botaoMudaLayout:active + .mural já não funciona mais -->
<header class="cabecalho">
    <h1 class="cabecalho-logo">Ceep </h1>
    <button id="botaoMudaLayout">Muda Layout</button>
</header>
<section class="mural">
    <div class="cartao"></div>
    <div class="cartao"></div>
</section>

```

Outro ponto é que enquanto o botão estiver sendo clicado, os cartões terão seu layout alterado. Porém, ao soltar o botão, voltamos para o layout inicial.

Mudanças permanentes de estilo como essa, após uma interação do usuário, ainda não são possíveis sem afetar nossa marcação. Para que nossos estilos se alterem em resposta a uma ação do usuário usaremos **JavaScript**.

Essa linguagem de programação roda no navegador e permite que façamos lógicas muito mais elaboradas sem a necessidade de um servidor. Além disso, ele possui ferramentas que nos permite interagir com os elementos do HTML, alterando seus estilos, conteúdo e atributos.

3.3 DOM: SUA PÁGINA NO MUNDO JAVASCRIPT

Para permitir alterações na página, ao carregar o HTML da página, os navegadores carregam em memória uma estrutura de dados que representa cada uma das nossas tags no JavaScript. Essa estrutura é chamada de **DOM (Document Object Model)**. Essa estrutura pode ser encontrada na propriedade global `document`.

O termo "documento" é frequentemente utilizado em referências à nossa página. No mundo front-end, documento e página são sinônimos.

Objetos Javascript

Mas como guardar tanta informação em apenas uma propriedade? Os principais tipos de dados do Javascript são `string`, `number` e `boolean`. Mas estes servem para apenas um valor, enquanto o `document` precisa guardar toda a estrutura da nossa página.

Sempre que pensamos em guardar diversos valores, a primeira solução que surge em nossa mente é um **array**. O array, permite guardar vários valores numa estrutura ordenada, dessa forma, se precisarmos do valor de volta, basta saber em que posição o colocamos.

```
//criando um array
var valores = [ 4 , 8.9 , true , "oi!" ]

//pegando o valor 4 que está primeira posição (posição zero):
valores[0];
```

Por causa dessa característica, o `document` poderia facilmente ser um array. O único problema é saber em qual posição está guardada a informação que queremos. Seria necessário algum tipo de documentação que listasse o número de todas as propriedades. O ideal seria que, ao invés de usarmos um número, usássemos o nome do que queremos. Assim, o Javascript criou um tipo de dado mais complicado que serve exatamente para isso, o **object**.

O **object** é um tipo de dado que permite guardar valores e dar nomes a eles. Depois usamos o nome para recuperá-lo:

```
//criando um objeto
var pessoa = {
    idade: 29,
    altura: 1.75,
    peso: 85,
    fuma: true,
    nome: "André"
}

//pegando o nome do objeto
pessoa.nome

//mudando a idade
pessoa.idade = 30
```

O valor guardado pode até ser uma `function`:

```
var pessoa = {
    idade: 29,
    altura: 1.75,
    peso: 85,
    fuma: true,
    nome: "André",
    calculaICM: function(){
        return altura * peso;
    }
}

pessoa.calculaICM()
```

O **DOM** encontrado na propriedade local `document` é uma estrutura que o navegador cria, onde ele representa **cada elemento do html como um object javascript**, com todas as suas propriedades (nome, id, classes...).

BOM

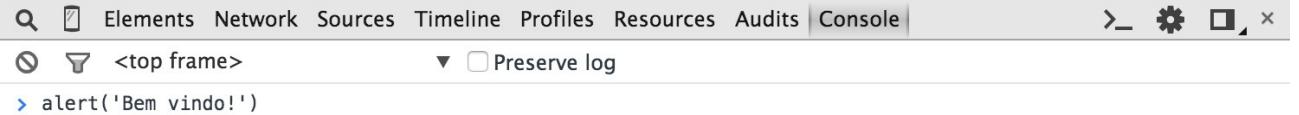
Além da estrutura da nossa página em memória, existe também uma estrutura que possui informações a respeito do próprio navegador, o BOM (Browser Object Model). Podemos acessá-lo através da propriedade **window**.

Também é nessa propriedade que ficam guardados todas as funções e variáveis globais, como as funções `alert` e `setTimeout`, e as variáveis `console` e até mesmo o `document`.

Alterações no DOM

Ao alterarmos esses objetos, o navegador sincroniza as mudanças e altera a aplicação em tempo real.

Para testar isso, novamente recorremos ao DevTools. Nele existe a opção **Console** onde podemos colocar códigos JS e ver os resultados.



The screenshot shows the DevTools interface with the 'Console' tab selected. The console log contains a single entry: `> alert('Bem vindo!')`. The output of this alert is not visible in the screenshot, but it would appear as a modal window above the browser window.

```
Elements Network Sources Timeline Profiles Resources Audits Console
✖
✖ <top frame> ▼  Preserve log
> alert('Bem vindo!')
```

querySelector

Como exemplo, vamos alterar a cor do cabeçalho da página. Precisamos primeiro pegar o elemento JavaScript que representa esse título:

```
document.querySelector("h1")
```

Esse comando usa os **seletores CSS** para encontrar os elementos na página. Usamos o seletor de `tagName`, mas poderíamos ter usado outros:

```
document.querySelector(".class")
document.querySelector("#id")
```

Executando no console, você vai perceber que o elemento correspondente é selecionado. Podemos então manipular seu conteúdo. Você pode ver o conteúdo textual dele com:

```
document.querySelector("h1").textContent
```

Essa propriedade inclusive pode receber valores e ser alterada:

```
var h1 = document.querySelector("h1").textContent
h1.textContent = "Novo título"
h1.id = "titulo"
```

QUERYSELECTORALL

Às vezes você precisa selecionar vários elementos na página. Várias tags com a classe `.secao` por exemplo. Se o retorno esperado é mais de um elemento, usamos `querySelectorAll` que devolve uma lista de elementos (array).

```
document.querySelectorAll(".cartao")
```

Podemos então acessar elementos nessa lista através da posição dele (começando em zero) e usando o colchetes:

```
// primeiro cartão
var cartoes = document.querySelectorAll(".cartao")
cartoes[0]
```

3.4 DESACOPLANDO O JAVASCRIPT DO CSS

A propriedade `style`

Para que o estilo do mural e dos cartões sejam alterados, podemos acessar a propriedade `style` que todo elemento HTML tem:

```
document.querySelector(".mural").style
```

A propriedade `style` tem, dentro dela, todas as propriedades de CSS que o browser conhece. Podemos acessar e alterar o valor do `flex-direction`, assim:

```
var mural = document.querySelector(".mural")
mural.style.flexDirection = "column"
```

Note que propriedades que contém hífen são escritas em camelCase.

Para alterar o estilo dos cartões, é preciso acessar cada um dos cartões e alterar seus estilos. Teremos que navegar por uma lista contendo todos os cartões.

```
var cartoes = document.querySelectorAll(".cartao")

for(var i = 0; i < cartoes.length; i++){
    cartoes[i].style.flexBasis = "auto";
    cartoes[i].style.width = "auto";
    cartoes[i].style.display = "block";
}
```

Quando alteramos estilos diretamente pelo javascript, estamos inserindo estilos inline no elemento. Exatamente como no exemplo abaixo:

```
<div class="mural" style="flex-direction:column">
<div class="cartao" style="flex-basis:auto; width:auto; display:block;></div>
<div class="cartao" style="flex-basis:auto; width:auto; display:block;></div>
```

```
</div>
```

O quanto fácil é manter os estilos do mural dessa forma? Se não quisermos mais usar flexbox, ou se quisermos aplicar uma transição na mudança de layout, em quais lugares teremos que mexer? No JavaScript. Dizemos que esse estilo está acoplado ao javascript.

Até agora nossos estilos foram aplicados aos nossos componentes, através das classes que colocávamos neles. Se nossa funcionalidade altera algum estilo, faremos com que ela modifique a classe do elemento, não seu `style`.

classList API

Para trabalhar com classes usamos a *ClassList API*. No caso do mural, no qual precisamos sobrescrever uma propriedade do CSS, adicionaremos uma classe:

```
document.querySelector(".mural").classList.add("mural--linhas")
```

No CSS:

```
.mural--linhas {  
    flex-direction: column;  
}  
  
.mural--linhas .cartao {  
    flex-direction: auto;  
    width: auto;  
    display: block;  
}
```

Há outras funções de manipulação possíveis. Podemos verificar se um elemento possui certa classe com `classList.contains()` recebendo o nome da classe.

```
var elemento = document.querySelector(".elemento")  
if(elemento.classList.contains("elemento")){  
    alert("Tem a classe")  
}
```

Ou ainda remover uma classe com certo nome usando `classList.remove()`. E, uma função bastante útil, podemos ligar/desligar uma classe alternadamente com `classList.toggle()`.

INDICANDO MODIFICADORES

É comum que classes sejam acrescentadas aos nossos componentes para alterar ou adicionar algum estilo em dado momento.

Essas classes não indicam a criação de componentes novos, mas a alteração de algum já criado. Para indicar isso, usaremos o seguinte padrão de nomenclatura:

```
.mural--linhas {  
    /* propriedades modificadas */  
}
```

Dizemos que `linhas` é um modificador da classe `mural`.

3.5 RELEMBRANDO EVENTOS JAVASCRIPT

A funcionalidade de troca de layout deve ser disparada pelo usuário. Isso significa ter algum elemento na tela clicável para disparar essa ação. Em HTML, o elemento semântico para disparar ações na página é o `button`:

```
<button id="mudaLayout">Linhas</button>
```

A renderização padrão é um botão que depois pode ser estilizado pelo CSS. Colocamos também um ID no elemento para poder referenciá-lo tanto nos estilos quanto no JS.

Mas o que esse botão faz? Nada. Um `button` perdido no HTML não dispara ação alguma. Para adicionar comportamento a ele, usaremos JavaScript. No caso, **Eventos JavaScript**.

Evento é o nome dado a alguma ação que pode ser disparada em algum elemento da página. Um botão, por exemplo, pode ser clicado, então possui o *evento de clique*. Mas não só. Poderíamos tratar o evento de passar o mouse (*mouseover*) em algum elemento. Ou observar quando o evento de scroll na página é disparado. Há muitos eventos diferentes e possíveis no JS.

OUTROS EVENTOS

O seguinte site tem uma relação de todos os eventos possíveis: <https://developer.mozilla.org/pt-BR/docs/Web/Events>

No nosso exemplo, queremos lidar com o evento de clique do botão. No JavaScript, isso significa **atrelar uma função** ao evento. Quando o clique acontecer, o navegador chama essa função pra gente. É o que chamamos de *função de callback*.

Para atrelar uma função de callback, podemos usar as propriedades das tags html relativas a eventos, que são as propriedades começadas com **on** mais o nome do evento. Nesse caso queremos atrelar uma função ao evento de **clique**, então usamos a propriedade **onclick** :

```
function alerta(){
    alert("Fui clicado!");
}

<button id="mudaLayout" onclick="alerta()">Linhas</button>
```

Esse código diz para o navegador que uma lógica javascript deve ser executada quando o evento de *click* acontecer. Por enquanto, apenas chamamos uma função que mostra um alerta.

3.6 JAVASCRIPT ONDE?

Acabamos de criar um botão que executa um código javascript no momento em que é clicado. Mas onde colocamos esse código javascript? Na propriedade **onclick**, **dentro do nosso html**

Mas html devia conter apenas conteúdo, e não lógica como fizemos. Felizmente, podemos resolver isso novamente com a ajuda do DOM. Nele podemos acessar qualquer propriedade das nossas tags e isso **inclui a propriedade onclick**. Ou seja, poderíamos fazer tudo pelo javascript.

```
document.querySelector("#mudaLayout").onclick = mudaLayout;
```

Note que ainda não queremos executar a função `mudaLayout`, queremos apenas guardar ela dentro da propriedade `onclick` para ser executada quando o evento for disparado. Por isso passamos o nome da função, sem o uso dos parênteses.

Mas e se precisarmos de dois comportamentos diferentes no clique desse botão? Toda vez que passamos uma nova função de callback para a propriedade `onclick`, nós estamos **apagando os outros callbacks que ele já possuía**. Isso significa que podemos até apagar comportamentos que são padrão no navegador, quebrando nossa página de maneiras imprevisíveis.

O ideal seria nós **adicionarmos uma nova** função de callback sem apagar as existentes. Para isso, foi criada a função `addEventListener`. Basta passar para ela o nome do evento e a função de callback:

```
document.querySelector("#mudaLayout").addEventListener("click", mudaLayout);
```

3.7 FUNÇÕES ANÔNIMAS

É muito comum que uma função cadastrada em um `addEventListener` não seja reaproveitada em nenhum lugar do código. Nesses casos, podemos criar uma função sem nome, que ninguém mais conseguirá chamar, como segundo parâmetro do `addEventListener`:

```
document.querySelector("#mudaLayout").addEventListener("click", function (){
    alert("Fui clicado!");
});
```

3.8 OUVINDO EVENTOS EM VÁRIOS ELEMENTOS

Antes de começar a inserir nossos cartões, vamos deixar pronta a funcionalidade de remoção deles. Para cada cartão teremos um botão de remover que quando clicado aplica a classe `cartao--some` no respectivo cartão:

```
.cartao--some {  
    opacity: 0;  
    transition: .2s ease-in;  
}
```

Diferentemente da funcionalidade anterior, nosso *Event Listener* será cadastrado em mais de um elemento da página. Para isso, precisaremos navegar numa lista com cada um.

```
var botoes = document.querySelectorAll(".opcoesDoCartao-remove");  
  
for(var i = 0; i < botoes.length; i++){  
    botoes[i].addEventListener("click", removeCartao);  
}
```

3.9 USANDO A ESTRUTURA DO DOM AO NOSSO FAVOR

Agora que conseguimos executar a `function removeCartao` sempre que o botão remover de cada cartão é clicado, precisamos implementar seu código.

A propriedade `this`

Primeiramente, precisamos saber qual foi o botão clicado.

```
function removeCartao(){  
    var botaoRemove = this;  
}
```

No javascript a propriedade `this` pode significar diversas coisas em lugares diferentes. No caso, dentro do callback de um *Event Listener*, ela aponta para o elemento no qual foi disparado o evento.

Navegando no DOM

Agora que temos o botão em mãos, podemos acessar qualquer outro elemento dentro dele com `botaoRemove.querySelector("seletor")`, porém, o que precisamos é acessar um de seus parentes, no caso, dois níveis acima.

```
<div class="cartao">  
    <div class="opcoesDoCartao">  
        <button class="opcoesDoCartao-remove">Remover</button>  
    </div>  
    <p class="cartao-conteudo">  
        Lorem...  
    </p>
```

```
</div>
```

O DOM é uma estrutura de dados complexa que armazena cada tag da página de forma hierárquica. Na computação, damos o nome de **árvore** para essa estrutura.

Dizemos que cada tag da página é um nó. Cada nó é interligado aos seus vizinhos. No nosso caso, precisamos acessar os pais do botão:

```
function removeCartao(){
    var botaoRemove = this;
    var cartao = botaoRemove.parentNode.parentNode;
}
```

A propriedade `parentNode` aponta sempre para o elemento pai do elemento.

Outras propriedades para navegação

Para acessar outros *Nodes* a partir de um elemento, podemos usar as seguintes propriedades:

- `parentNode` - Tag pai do elemento
- `previousSibling` - Tag irmã que veio antes do elemento
- `nextSibling` - Tag irmã que veio depois do elemento
- `childNodes` - Lista com todos os nós filhos
- `firstChild` - Primeiro nó filho do elemento
- `lastChild` - Último nó filho do elemento

3.10 DESACOPLANDO NOSSO CÓDIGO DA ESTRUTURA

Ao utilizar a propriedade `parentNode` estamos dizendo que nossa funcionalidade depende do posicionamento do botão dentro do cartão. Se em algum momento decidirmos não colocar o botão de remoção dentro da `<div class="opcoesDoCartao">`, nossa lógica de remoção para de funcionar.

Para que isso não aconteça, precisamos ligar o botão ao cartão de forma independente da estrutura do DOM. O botão precisa apontar diretamente para o cartão.

Qualquer elemento na página pode ser identificado com um `id`.

```
<div class="cartao" id="cartao_1">
</div>
```

É possível selecioná-lo dentro do DOM com um `document.querySelector("#cartao_1")`.

Mas como ter acesso ao id do cartão cujo botão de remoção está sendo clicado? Podemos seguir a ideia dos links que apontam para elementos dentro da mesma página.

```
<a href="#cartao_1">Link para o cartão</a>
```

Não queremos o comportamento de um `<a>`, mas queremos um atributo que aponte para o cartão,

um atributo que diz qual o cartão a ser removido.

```
<div class="cartao">
  <div class="opcoesDoCartao" id="cartao_1">
    <button class="opcoesDoCartao-remove" href="#cartao_1">Remover</button>
  </div>
  <p class="cartao-conteudo">
    Lorem...
  </p>
</div>
```

Podemos acessar qualquer atributo dos elementos dentro do DOM.

Acessando outros atributos das tags

Cada elemento HTML possui diversos atributos para as mais variadas tarefas. Você pode inclusive acessar o valor desses atributos no JavaScript com `getAttribute`. Por exemplo:

```
<a href="#cartao_1">Link para o cartão</a>
```

Podemos acessar o valor do atributo `href` no JS para efetuar alguma operação:

```
var a = document.querySelector("a");
var href = a.getAttribute("href");

console.log(href);
```

Essa possibilidade de comunicação entre o HTML e o JavaScript é muito útil em diversos cenários. Tão útil que é bastante frequente querermos pendurar todo tipo de dados no HTML para depois acessar pelo JS.

Há quem use o `class` para isso, por exemplo. Usando valores especiais lá dentro que depois pode acessar via `classList` no JS. Mas não é uma boa ideia. Além de misturar as coisas, as classes são limitadas (o espaço é um separador, então não permite o uso de valores com espaço).

Atributos customizados no DOM

No caso do nosso botão de remoção, adicionamos o atributo `href` à tag, um atributo que não faz parte da especificação de um `<button>`.

Adicionar atributos aleatoriamente nas tags não é o que recomenda a especificação. Se precisamos criar atributos para guardar informações para nossas lógicas podemos nos aproveitar de uma nova categoria de atributos nas tags chamada de **Data Attributes**. São atributos que podem ter qualquer nome mas possuem prefixo `data-`. Por exemplo:

```
<button class="opcoesDoCartao-remove" data-ref="cartao_1">Remover</button>
```

No código anterior, adicionamos um atributo customizado `data-ref`. Ele é um atributo nosso, sem nenhum valor semântico nem nenhuma funcionalidade especial. Colocá-lo apenas no HTML não vai fazer nada. Precisamos acessá-lo e implementar nossa funcionalidade.

```

function removeCartao(){
    var botaoRemove = this;
    var idDoCartao = botaoRemove.getAttribute("data-ref");
    var cartao = document.querySelector("#" + idDoCartao);

    cartao.classList.add('cartao--some');
}

```

Repare que acessamos o atributo especial também através de `getAttribute`.

A classe "cartao--some" aplicará um `opacity: 0` no cartão.

dataset API

Existe também uma forma especial de acessar os data attributes com mais facilidade, através da propriedade `dataset`. Por exemplo, para acessar o `data-ref` faríamos:

```

<button class="opcoesDoCartao-remove" data-ref="cartao_1">Remover</button>

document.querySelector("button").dataset.ref;

```

Podemos também atribuir um valor e mudar o atributo:

```
document.querySelector("button").dataset.ref = "cartao_1";
```

3.11 REMOVENDO ELEMENTOS DO DOM E RELEMBRANDO SETTIMEOUT

Colocar um `opacity: 0` não é o suficiente para nossa função de remover o cartão. É preciso realmente remover o elemento da página. Para isso, usamos a função `remove` no nosso objeto:

```

function removeCartao(){
    var botaoRemove = this;
    var seletorCartao = "#cartao_" + botaoRemove.dataset.ref;
    var cartao = document.querySelector(seletorCartao);

    cartao.classList.add("cartao--some");
    cartao.remove();
}

```

No caso, nossa transição de `.2s` aplicada na classe `cartao--some` não será visível, já que, logo após a adição da classe, o cartão é removido.

Precisamos esperar que a transição acabe antes de remover o cartão. Para isso, criaremos um *timer* que executará nosso trecho de código após um certo tempo.

A função `setTimeout` permite que agendemos alguma função para execução no futuro e recebe o nome da função a ser executada e o número de milissegundos a esperar.

```

function removeCartao(){
    var botaoRemove = this;
    var seletorCartao = "#cartao_" + botaoRemove.dataset.ref;
    var cartao = document.querySelector(seletorCartao);

```

```
cartao.classList.add("cartao--some");

setTimeout(function(){
    cartao.remove()
}, 170);
}
```

É uma função útil para implementar funcionalidades que devem esperar pra executar.

SETINTERVAL

O `setInterval` é muito parecido com o `setTimeout`. A única diferença é que a função passada como parâmetro será chamada recorrentemente no intervalo de tempo definido no segundo parâmetro da função.

3.12 CSS3 TRANSITIONS

Com as transitions, conseguimos animar o processo de mudança de algum valor do CSS.

Por exemplo: temos um elemento na posição `top:10px` e, quando passarmos o mouse em cima (`hover`), queremos que o elemento vá para `top:30px`. O CSS básico é:

```
#teste {
    position: relative;
    top: 0;
}
#teste:hover {
    top: 30px;
}
```

Isso funciona, mas o elemento é deslocado de uma vez quando passamos o mouse. E se quisermos algo mais sutil? Uma animação desse valor mudando lentamente, mostrando o elemento se deslocando na tela? Usamos CSS3 Transitions.

Sua sintaxe possui vários recursos mas seu uso mais simples, para esse nosso caso, seria apenas:

```
#teste:hover {
    transition: top 2s;
}
```

Isso indica que queremos animar a propriedade `top` durante 2 segundos.

Por padrão, a animação é linear, mas temos outros tipos para animações mais suaves:

- `linear` - velocidade constante na animação;
- `ease` - redução gradual na velocidade da animação;
- `ease-in` - aumento gradual na velocidade da animação;
- `ease-in-out` - aumento gradual, depois redução gradual na velocidade da animação;

- `cubic-bezier(x1, y1, x2, y2)` - curva de velocidade para animação customizada (avançado);

```
#teste:hover {
  transition: top 2s ease;
}
```

Para explorar o comportamento dos tipos de animações disponíveis, e como criar uma curva de velocidade customizada para sua animação, existe uma ferramenta que auxilia a criação do `cubic-bezier` : <http://www.roblapla.com/examples/bezierBuilder/>

Podemos ainda usar mais de uma propriedade ao mesmo tempo, incluindo cores!

```
#teste {
  position: relative;
  top: 0;
  color: white;
}
#teste:hover {
  top: 30px;
  color: red;
  transition: top 2s, color 1s ease;
}
```

Se quisermos a mesma animação, mesmo tempo, mesmo efeito para todas as propriedades, podemos usar o atalho `all` (que já é o valor padrão, inclusive):

```
#teste:hover {
  transition: all 2s ease;
}
```

Essa especificação, ainda em estágio inicial, é suportada em todos os navegadores modernos, incluindo o IE 10. Mas precisamos de prefixos em vários browsers.

```
#teste:hover {
  -webkit-transition: all 2s ease;
  -moz-transition: all 2s ease;
  -o-transition: all 2s ease;
  transition: all 2s ease;
}
```

JQUERY

O jQuery é o **mais usado framework JavaScript** do mercado. Ele é muito importante no dia a dia e todo programador front-end deve estar pelo menos familiarizado com ele.

O jQuery não é um bicho de sete cabeças. Pense nele como um conjunto de funções JavaScript que você importa na sua página. São funções que te ajudam, por exemplo, a esconder as diferenças entre os navegadores.

Muito mais do que apenas garantir a compatibilidade do seu código

O jQuery, além de "blindar" o programador das diferenças entre navegadores, nos permite **fazer mais com menos código**, o que é muito bem-vindo em nosso projeto.

Tudo o que aprendemos em JavaScript é aplicável ao jQuery, já que ele é escrito utilizando esta linguagem, o que muda é a maneira pela qual **acessamos, modificamos e navegamos** pela estrutura do documento com ele.

4.1 CONHECENDO O JQUERY

Para utilizarmos o jQuery, o primeiro passo é importá-lo em nossa página:

```
<script src="js/lib/jquery.js"></script>
```

O jQuery foi feito para atender ao **padrão comum de programação front-end**, onde primeiro obtemos um objeto da página e depois utilizamos diversas funções e atributos para modificar seu estado, e consequentemente, sua exibição no navegador.

Sendo assim, veremos como o jQuery acessa e altera elementos, inclusive como funciona seu mecanismo de evento.

jQuery object

O primeiro componente que vamos analisar, e o principal, é o **jQuery object**, que no código também é conhecido como `$`, seu alias. O uso mais comum do `$` é como a função de fábrica de objetos do jQuery:

```
var jsBotao = document.querySelector("#mudaLayout"); // com JavaScript puro
```

```
var jqBotao = $("#mudaLayout"); // com jQuery. Bem menor, não?
```

No exemplo acima, ao ser executada, a função `$` recebe uma String como argumento, contendo um seletor CSS. O retorno é um objeto assim como quando utilizamos a função `querySelector` de `document`, mas com uma pequena diferença: ele não é o elemento do DOM, mas um objeto do jQuery que guarda o elemento do DOM.

VARIÁVEL "JQUERY"

No lugar de usar o recomendado alias `$`, você pode usar diretamente o objeto `jQuery`, mas a primeira forma é recomendada, pela simplicidade da escrita:

```
var jqBotao = jQuery("#mudaLayout");
```

Outra vantagem do uso do jQuery para selecionar os elementos do documento é que suas funções já fazem o tratamento necessário para que recebam um elemento ou um objeto que contenha mais de um elemento.

```
// usamos querySelectorAll, porque queremos uma lista de Li's
var jsCartoes = document.querySelectorAll(".cartao");

// usamos a mesma função $, só que agora ela traz uma lista de elementos.
var jqCartoes = $(".cartao");
```

JQUERY E SELETORES CSS

Mais uma vez é necessário conhecer os seletores CSS, pois o jQuery suporta todos, inclusive os seletores mais modernos do CSS3. Além disso, como veremos mais a frente, o jQuery possui seletores exclusivos. Não se preocupe com eles ainda, veremos algum deles mais a frente.

jQuery Object "blinda" o programador

Este elemento funciona como uma "blindagem", já que agora não manipulamos diretamente o elemento do DOM. Qualquer alteração ou adição de evento deve passar por ele e **tudo é feito através de funções**. É aí que mora a "mágica" da compatibilidade, pois essas funções são crossbrowser, funcionando nos mais diversos navegadores.

Um exemplo que ilustra isso é alteração do texto de um elemento, em nosso exemplo, o botão de mudar o layout. Com JavaScript puro, manipulávamos diretamente a propriedade `textContent` do elemento:

```
var botao = document.querySelector("#mudaLayout");
botao.textContent = "Blocos";
```

Uma alteração inocente como essa só funciona a partir do Internet Explorer 10. Nas versões anteriores deste navegador é necessário usar a propriedade `innerText`.

Com jQuery, não nos preocupamos se a propriedade existe ou não, pedimos à sua função `text` que execute esta tarefa para nós:

```
var botao = $("#mudaLayout");
botao.text("Blocos");
```

Como já foi dito antes, as funções do jQuery se encarregarão da compatibilidade do nosso código entre múltiplos navegadores.

Manipulando múltiplos elementos com jQuery

Uma forma interessante de entender a diferença entre o *jQuery object* e um *elemento do DOM* é a manipulação de múltiplos elementos. Com o jQuery não é necessário realizar um `for` para mexer em múltiplos elementos.

```
<ul>
  <li class="item">Item 1</li>
  <li class="item">Item 2</li>
  <li class="item">Item 3</li>
</ul>

var itens = $(".item");
itens.text("Sou um item");
```

No exemplo acima, com uma única chamada, todas as li's receberão "Sou um item". Isso porque o jQuery object que está na variável `itens` guarda dentro dele **todos os elementos** que tem a classe "item". Queremos escrever menos código e o jQuery vai nos ajudar muito nisso.

E você ainda pode evitar a declaração da variável `itens`:

```
$(".item").text("Sou um item");
```

O padrão acima é bastante utilizado.

E A FUNÇÃO `VAL()`?

A função `val()` é utilizada para obter e alterar o valor de elementos como `input`, `select` e `textarea`.

Padrão de utilização

Um padrão para se utilizar o jQuery é o de escrever todo código em uma função anônima e enviá-la com argumento para a função `$`:

```
$(function() {
```

```
var jqBotao = $("#botaoaviso");
jqBotao.text("novo texto");
});
```

A forma acima é um atalho para a função `ready` :

```
$(document).ready(function() {
  var jqBotao = $("#botaoaviso");
  jqBotao.text("novo texto");
});
```

A principal vantagem desse padrão é que, como estamos interagindo com elementos do documento, essencialmente, o jQuery espera todos os elementos serem carregados e a página disparar o evento "DOMContentLoaded", garantindo assim que nosso código vai funcionar sem deixar nenhum elemento "para trás".

Esse padrão de utilização é que permite importar scripts que manipulem o documento na tag `<head>`. No entanto, ele é desnecessário se for seguida a boa prática de carregar os scripts antes do fechamento da tag **body**.

4.2 EVENTOS

Para criar um cartão, nosso usuário digitará o conteúdo dele num `<textarea>`, dentro de um formulário.

```
<form class="novoCartao" action="#">
  <textarea class="novoCartao-conteudo"></textarea>
  <input type="submit" value="Adicionar">
</form>
```

No momento em que o formulário for enviado, devemos criar o cartão.

O uso do jQuery para a adição de funções atribuídas a eventos é bem parecida com o do JavaScript puro, porém a função chama-se `on()`.

```
var form = $(".novoCartao");
form.on("submit", function() {
  //aqui o código de criação do cartão
});
```

No lugar de jogarmos o resultado da busca numa variável, podemos associar o elemento de maneira **inline** :

```
$(".novoCartao").on("submit", function() {
  //aqui o código de criação do cartão
});
```

O comportamento padrão do formulário é redirecionar o usuário para o endereço indicado no atributo `action`. Se não declaramos um `action` ele simplesmente recarrega a página. Assim, qualquer cartão que criarmos será perdido. Para isso não acontecer, queremos cancelar esse comportamento default.

Cancelando os eventos

Em qualquer evento, temos disponível no callback do nosso *Event Listener* uma variável `event`. Para ter acesso a ela, precisamos pedi-la como parâmetro:

```
$(".novoCartao").on("submit", function(event) {  
    //aqui o código de criação do cartão  
});
```

Com o `event` em mãos, podemos cancelar o evento de submit da seguinte maneira:

```
$(".novoCartao").on("submit", function(event) {  
    event.preventDefault();  
    //aqui o código de criação do cartão  
});
```

Ouvindo vários eventos

A função `on()` é tão versátil que pode receber outros eventos ao mesmo tempo em que aproveita a mesma função passada como parâmetro:

```
$("#button").on("click mouseover", function(event) {  
    alert("Executou");  
    event.preventDefault();  
});
```

Outro ponto é que não precisamos nos preocupar se o navegador suporta `addEventListener` ou `attachEvent` pois a função `on()` se encarregará desta tarefa para nós.

E A FUNÇÃO BIND?

Nas versões do jQuery anteriores à 1.7, a função `bind` era utilizada mas, a partir da versão 1.7, recomenda-se o uso da função `on`. A sintaxe é a mesma, o que muda é o nome da função:

```
$("#button").bind("click", function(event) {  
    alert("Executou");  
    event.preventDefault();  
});
```

Shorthand Event

Também é possível utilizar os métodos de atalho, os **shorthand events**, com jQuery para os eventos mais comuns do JavaScript. Temos, por exemplo, a função `submit`, atalho para o `on("submit", ...)`:

```
$(".novoCartao").submit(function(event) {  
    //aqui o código de criação do cartão  
});
```

Veja que escrevemos menos código usando os *shorthand events* do jQuery. Assim, vamos usar `on()` quando formos executar um mesmo código para diferentes eventos ou quando formos trabalhar com o

mecanismo de delegação de eventos, assunto que veremos mais a frente.

4.3 NAVEGAÇÃO NO DOM COM JQUERY

Da mesma maneira que o JavaScript puro permite navegar pela árvore do DOM, podemos fazer a mesma coisa com jQuery, mas sem depender tanto da estrutura do documento. As funções mais utilizadas são:

- **next(seletor)**: retorna o irmão do elemento atual se ele bater com o seletor dado. Quando o seletor é omitido, sempre retorna o irmão.
- **nextAll(seletor)**: retorna os próximos elementos irmãos do atual, desde que eles batam com o seletor. Se o seletor for omitido, retorna todos os irmãos a partir do elemento.
- **prev(seletor)**: idêntico ao `next`, mas pega o irmão anterior.
- **prevAll(seletor)**: idêntico ao `nextAll`, mas pega os irmãos anteriores.
- **parent(seletor)**: retorna o pai imediatamente acima do elemento atual se ele bater com o seletor dado. Quando o seletor é omitido, sempre retorna o pai.
- **parents(seletor)**: retorna todos os elementos pais do elemento atual até o topo da árvore do DOM que batam com o seletor. Quando o seletor é omitido, retorna todos os elementos pais.
- **closest(seletor)**: sobe na hierarquia retornando o primeiro elemento ancestral do atual que bate com o seletor.

4.4 MODIFICANDO O DOM COM JQUERY

O jQuery traz uma série de facilidades para modificarmos a árvore do DOM que veremos ao longo dos capítulos. Por exemplo, é possível remover um elemento através da função `remove`:

```
$(".cartao").remove(); // remove os cartões
```

Encadeamento de funções

O jQuery permite o encadeamento de funções. Abaixo, a versão não encadeada:

```
var botao = $(".opcoesDoCartao-remove");
botao.text("Remover");
botao.click(function(event) {
    // seu código
})
```

Podemos escrever o mesmo código, só que encadeando as chamadas de funções a partir de um objeto:

```
$(".opcoesDoCartao-remove").text("Remover").click(function(event) {
    // seu código
});
```

4.5 FUNÇÕES MAIS COMUNS DO JQUERY

A seguir, algumas funções muito usadas quando trabalhamos com jQuery.

CSS

Podemos alterar o estilo de elementos dinamicamente com jQuery através de sua função **css**:

```
$(".mural").css("flex-direction", "column").css("background-color", "black");
```

Repare que, no exemplo acima, chamamos duas vezes a função **css**. Podemos evitar isso passando como parâmetro um objeto do JavaScript onde cada propriedade equivale a uma propriedade do CSS. A diferença é que a propriedade vem entre aspas:

```
var estilos = {  
    "flex-direction" : "column",  
    "background-color" : "black"  
}  
$(".aviso").css(estilos);
```

addClass e removeClass

Como no JavaScript puro, o ideal é deixarmos o CSS no seu 'quadrado', facilitando sua manutenção. Podemos adicionar e remover classes usando jQuery com suas funções **addClass** e **removeClass** respectivamente:

```
$(".aviso").addClass("invisivel");  
$(".aviso").removeClass("invisivel");
```

toggleClass

Quando queremos 'ligar' ou 'desligar' uma classe, podemos ainda utilizar a função **toggleClass**:

```
// se não tiver a classe, coloca; se tiver, remove  
$(".aviso").toggleClass("invisivel");
```

show e hide

No exemplo acima, usamos uma classe que torna elementos da nossa página invisíveis mas, com jQuery, podemos usar suas funções **show** e **hide** para conseguir a mesma funcionalidade:

```
$(".aviso").hide(); // esconde  
$(".aviso").show(); // mostra
```

toggle

Ocultar e exibir elementos na tela é algo muito comum. É por isso que existe a função **toggle**. Quando chamada num elemento visível, ela o tornará invisível. Quando chamada mais uma vez, tornará o elemento visível novamente:

```
// se estiver visível, torna invisível;
```

```
// se estiver invisível, torna visível
$(".aviso").toggle();
```

Agora que avançamos mais um pouco no uso do jQuery, vamos ver quais dessas funções podem nos ajudar a fazer a criação dos nossos cartões.

4.6 CONSTRUINDO ELEMENTOS COM JQUERY

Quando o usuário submete o formulário, precisamos que um novo cartão apareça na página. Bom, sabemos que no javascript todos os nossos cartões são elementos do DOM. Portanto, precisamos **criar um novo elemento**. Para isso existe no objeto `document` a função `createElement`

```
var novoCartão = document.createElement("div");
novoCartão.classList.add("cartao");
```

O resultado desse código seria:

```
<div class="cartao"></div>
```

Dessa forma, criamos um elemento do DOM na mão. Portanto, perdemos as vantagens do jQuery. Por isso, o jQuery tem sua própria forma de criar elementos.

Novos elementos com a função \$

A função `$` também pode ser utilizada para criar elementos, assim como a função `createElement`, porém com o uso de menos código. Para isso, basta que seu primeiro argumento seja uma string que seja uma tag válida do HTML:

```
var botao = $("<button>");
```

Para manipular esse objeto, podemos utilizar várias funções, tanto do JavaScript como do jQuery, mas podemos definir os atributos e valores do elemento em sua construção.

Alterando propriedades

O resultado da execução do código acima seria uma tag, como exemplificada abaixo:

```
<button></button>
```

No exemplo acima, nosso botão carece de atributos. Podemos adicioná-los através da função `attr`:

```
var botao = $("<button>");
botao.attr("data-ref", "cartao_1");
botao.text("Remover");
```

Ou, se você preferir, utilizando o recomendado encadeamento de funções:

```
var botao = $("<button>").attr("data-ref", "cartao_1").text("Remover");
```

O resultado da execução do código acima seria uma tag como exemplificada abaixo:

```
<button data-ref="cartao_1">Remover</button>
```

Podemos chegar no mesmo resultado passando para a função \$ um objeto JavaScript contendo os atributos que queremos definir no novo elemento:

```
var botao = $("<button>", {data-ref: "cartao_1"}).text("Remover");
```

O resultado do código acima é exatamente igual à versão anterior.

Agora que criamos o elemento, assim como utilizando somente JavaScript puro, ele só existe em memória, não faz parte do documento. Precisamos utilizar alguma função para colocá-lo no documento.

Incluindo elementos no documento

O jQuery tem diversas funções que nos permitem adicionar elementos no documento utilizando qualquer elemento como referência. A seguir, vamos explorar alguns exemplos considerando a seguinte estrutura HTML:

```
<ul id="menu">
  <li class="item">
    <a href="#">Home</a>
  </li>
  <li class="item">
    <a href="#">Empresa</a>
  </li>
  <li class="item">
    <a href="#">Produtos</a>
  </li>
</ul>
```

appendTo() e prependTo()

A função **appendTo** inclui o elemento como **último filho (child)** dos elementos que atendem o seletor:

```
$(<span>, { class : "info" }).text("Novo span!").appendTo(".item");
```

No exemplo acima, a função **appendTo** coloca um **** como último filho de cada elemento que receber a classe "item" no documento:

```
<ul id="menu">
  <li class="item">
    <a href="#">Home</a>
    <span class="info">Novo span!</span>
  </li>
  <li class="item">
    <a href="#">Empresa</a>
    <span class="info">Novo span!</span>
  </li>
  <li class="item">
    <a href="#">Produtos</a>
    <span class="info">Novo span!</span>
  </li>
</ul>
```

Similar ao `appendTo` existe o `prependTo`, que coloca o elemento como **primeiro filho** dos elementos obtidos pelo seletor.

```
$( "<span>" , { class : "info" }).text("Novo span!").prependTo(".item");  
  
<ul id="menu">  
  <li class="item">  
    <span class="info">Novo span!</span>  
    <a href="#">Home</a>  
  </li>  
  <li class="item">  
    <span class="info">Novo span!</span>  
    <a href="#">Empresa</a>  
  </li>  
  <li class="item">  
    <span class="info">Novo span!</span>  
    <a href="#">Produtos</a>  
  </li>  
</ul>
```

APPEND() E PREPEND()

Existe também as funções `append` e `prepend`. Ambas são equivalentes a `appendTo` e a `prependTo` respectivamente. A diferença é que partimos de onde queremos incluir:

```
var elemento = $("<span>" , { class : "info" }).text("Novo span!");  
$(".item").append(elemento);  
$(".item").prepend(elemento);
```

insertAfter()

A função `insertAfter` adiciona o elemento como **irmão (sibling)** logo após cada elemento retornado pelo seletor:

```
// Nesse exemplo adicionei um texto ao elemento  
var novoSpan = $("<span>" , { class : "info" }).text("Novo span!");  
  
$(novoSpan).insertAfter("#menu li.item a");  
  
<ul id="menu">  
  <li class="item">  
    <a href="#">Home</a>  
    <span class="info">Novo span!</span>  
  </li>  
  <li class="item">  
    <a href="#">Empresa</a>  
    <span class="info">Novo span!</span>  
  </li>  
  <li class="item">  
    <a href="#">Produtos</a>  
    <span class="info">Novo span!</span>  
  </li>  
</ul>
```

insertBefore()

A função `insertBefore` adiciona o elemento como **irmão (sibling)** antes de cada elemento retornado pelo seletor:

```
// Nesse exemplo adicionei um texto ao elemento
var novoSpan = $("<span>", { class : "info" }).text("Novo span!");
$(novoSpan).insertBefore("#menu li.item a");
```

O resultado seria:

```
<ul id="menu">
  <li class="item">
    <span class="info">Novo span!</span>
    <a href="#">Home</a>
  </li>
  <li class="item">
    <span class="info">Novo span!</span>
    <a href="#">Empresa</a>
  </li>
  <li class="item">
    <span class="info">Novo span!</span>
    <a href="#">Produtos</a>
  </li>
</ul>
```

AFTER() E BEFORE()

Também existem as funções `after()` e `before()` que são equivalentes às funções `insertAfter` e `insertBefore`. A diferença é que partimos da referência do documento:

```
var novoSpan = $("<span>", { class : "info" }).text("Novo span!");
$("#menu li.item a").after(novoSpan);
$("#menu li.item a").before(novoSpan);
```

AJAX E A VIDA ASSÍNCRONA

5.1 AJAX COM JQUERY

Um dos objetos mais poderosos do JavaScript é o objeto XMLHttpRequest. Esse objeto é capaz de disparar uma requisição HTTP e nos fornece acesso às informações retornadas como resposta. O objeto XMLHttpRequest mudou a maneira com que desenvolvemos aplicações para a Internet, principalmente aquelas que requerem muitas interações do usuário com a página.

Hoje em dia é comum as aplicações, em seu lado do servidor, estarem preparadas para responder a determinada requisição com uma porção menor de informações em um fragmento de HTML ou até em um outro formato de serialização, como XML (formato para o qual foi pensado o objeto XMLHttpRequest), ou até mesmo um objeto do JavaScript (JSON, JavaScript Object Notation).

Essa requisição feita pelo XMLHttpRequest e a mudança no documento que realizamos a partir da resposta dessa requisição é o que chamamos de AJAX (Asynchronous JavaScript And XML). Com essa técnica, conseguimos alterar apenas uma parte de uma página sem ter que fazer o navegador recarregar a página inteira, evitando o tráfego desnecessário de informações.

Internet Explorer

Um dos problemas com a técnica AJAX é que versões muito antigas do Internet Explorer não implementam esse objeto em seu interpretador de JavaScript. Em vez disso, é necessário utilizar um componente ActiveX, que funciona de maneira semelhante, porém com sintaxe um tanto diferente.

A função \$.ajax

Já vimos que um dos objetivos do jQuery é fornecer uma maneira de escrevermos um único script e a biblioteca se encarregar de selecionar, a partir das funcionalidades disponíveis em cada navegador, qual objeto ou componente deve ser utilizado. Para fazer chamadas assíncronas com o jQuery podemos utilizar seu utilitário `ajax` :

```
$.ajax(/* objeto com configurações */);
```

Esse *objeto com configurações* que passamos como argumento para o utilitário `ajax` deve conter pelo menos dois atributos: o endereço para onde será feita a requisição sob a chave `url` e uma função que é executada caso a requisição retorne com sucesso sob a chave `success` :

```

$.ajax({
  url: "http://www.servidor.com/servico",
  success: function (data, textStatus, jqXHR) {
    // a resposta da requisição pode ser acessada pelo objeto "data"
  }
});

```

Passagem de parâmetros

Caso seja necessário passarmos alguma informação na requisição, como parâmetro, podemos incluí-las na URL, por exemplo `http://www.servidor.com/servico?promocao=janeiro`. Note que os parâmetros de requisição devem ser convertidos para um formato possível de ser transmitido em uma URL.

Para o trabalho de converter strings em valores válidos para uma URL (a função `encodeURIComponent("string qualquer")` faz isso para nós), podemos passar todas as informações de parâmetro num objeto `data` para o utilitário `ajax`:

```

$.ajax({
  url: "http://www.servidor.com/servico",
  data: {
    "busca": "evento caelum"
  },
  success: function (data, textStatus, jqXHR) {
    // callback
  }
});

```

Formato da resposta

Como mencionado antes, uma requisição AJAX pode devolver uma resposta em diversos formatos: HTML, XML, JSON etc. A função `ajax` do jQuery permite especificarmos qual será o formato da resposta do servidor para que o próprio jQuery processe a resposta e nos dê um objeto do JavaScript que representa essa resposta. Para isso, basta passarmos uma propriedade `dataType` no objeto de configurações que passamos para essa função.

Se a resposta for um XML, por exemplo, podemos chamar a função `ajax` da seguinte forma:

```

$.ajax({
  url: "http://www.servidor.com/servico",
  data: {
    "busca": "evento caelum"
  },
  dataType: "xml",
  success: function (data, textStatus, jqXHR) {
    // data será um objeto do JavaScript que representa o XML
  }
});

```

Assim, com esse utilitário é possível obter uma porção de informações relevantes, de uma porção de serviços úteis.

GOOGLE CHROME E ACESSO A ARQUIVOS LOCAIS

O Google Chrome não permite requisições AJAX localmente por questões de segurança. Para permitir que uma página local chame outra utilizando AJAX no Chrome é necessário iniciar o navegador pela linha de comando passando parâmetros especiais:

Linux: `google-chrome --allow-file-access-from-files` Mac: `open /Applications/Google\ Chrome.app --args --allow-file-access-from-files`

Lidando com erros em requisições AJAX

Nem sempre as coisas saem conforme o esperado, principalmente em um ambiente tão heterogêneo como a web. A conexão pode cair, o servidor pode estar fora do ar e inúmeras outras coisas podem acontecer. Como lidar com esses problemas quando estivermos trabalhando com requisições AJAX?

A função `$.ajax` e todas as suas especializações permitem a adição de um *error handler* por meio da função `fail` que, em sua função de callback, recebe uma referência ao objeto XMLHttpRequest, que contém informações de erros como `status` e `responseText`.

```
$.ajax( ..... ).fail(function(xhr){
  console.log("erro( " + xhr.status + "):" + xhr.responseText);
});
```

O exemplo acima mostra apenas o status do erro e o texto recebido como resposta mas, em outros cenários, esta função poderia avisar elegantemente ao usuário sobre a ocorrência de algum problema ou definir um comportamento padrão a ser executado em caso de falha.

5.2 JSON - JAVASCRIPT OBJECT NOTATION

JSON (JavaScript Object Notation) é um formato de troca de dados. A sua estrutura facilita sua manipulação e criação, tanto para seres humanos quanto para máquinas. Há ganho de performance em seu processamento se comparado com o XML. Este formato nada mais é do que um objeto JavaScript, com a diferença de seus atributos virem sempre entre aspas.

Um exemplo de JSON:

```
{
  "nome": "Golden Valve",
  "preco": 109.0,
  "descricao": "Feito com os mais finos dos materiais especialmente para você",
  "imagem": "img/produto-6.jpg"
}
```

5.3 \$.GETJSON

Apesar do AJAX ser uma técnica que, desde a sua concepção, se preocupa com arquivos XML, é muito comum servidores enviarem um JSON (JavaScript Object Notation) como resposta às requisições.

O JSON é muito menor do que um XML e o efeito mais direto disto é a rapidez com que os dados são transferidos, sem falar da rapidez com a qual o browser realiza o processamento dos dados.

O jQuery possui a função `$.getJSON`, especializada em trabalhar com dados no formato JSON:

```
$.getJSON("http://www.servidor.com/servico", function(retorno) {  
});
```

O primeiro parâmetro é a URL que retornará os dados; o segundo, a função de `callback` que recebe como parâmetro os dados retornado pela URL.

5.4 SAME ORIGIN POLICY E CORS

Quando o utilitário `ajax` do jQuery faz uma requisição, assim como qualquer requisição com o objeto XMLHttpRequest, ou o componente ActiveX correspondente para o Internet Explorer, espera-se que essa requisição seja feita a partir do mesmo domínio, ou seja, que a página que deseja fazer uso dessas informações faça parte da mesma aplicação.

Essa é uma medida de segurança chamada de "política de mesma origem" (*same origin policy*). Até mesmo requisições para o mesmo endereço de domínio, mas com protocolo diferente (por exemplo a página servida pelo protocolo http requisitando uma URL https), ou porta diferente são rejeitadas.

Essa segurança é muito importante na Web, mas é limitadora. Como fazer uma página que integre serviços externos de terceiros? Ou mesmo chamar um serviço nosso mas que, por algum motivo, precise estar em um domínio ou subdomínio diferente?

CORS - Cross Origin Resource Sharing

A solução definitiva para essa questão é o CORS (Cross Origin Resource Sharing) que foi adicionado ao JavaScript para permitir requisições AJAX cross-domain.

Ele é uma extensão ao padrão XHR (XMLHttpRequest) e funciona realizando um tipo de handshaking (preflighting) pedindo autorização ao outro servidor.

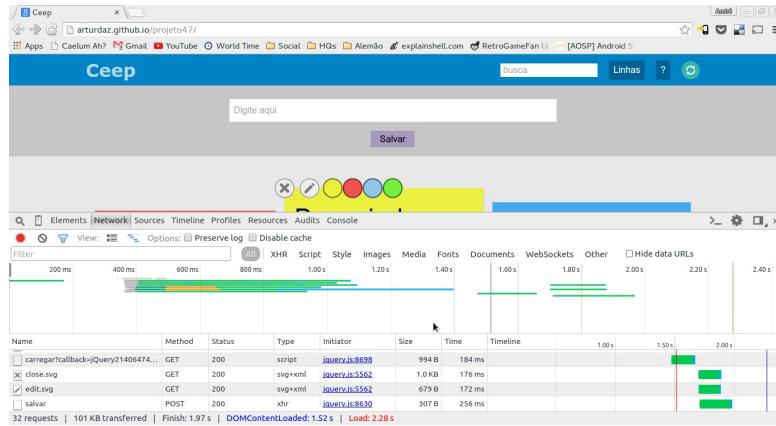
Na verdade é um novo cabeçalho HTTP que deve ser enviado pelo servidor para autorizar o uso do serviço pelo JavaScript. O navegador checa a presença desse header quando fazemos nossa chamada AJAX para ver se podemos ler a resposta do serviço.

O cabeçalho é o **Access-Control-Allow-Origin** que recebe uma lista de domínios autorizados a chamar aquele serviço. Podemos autorizar todos os domínios e deixar o serviço público com **Access-Control-Allow-Origin: ***.

Nosso exercício anterior funcionou porque o servidor foi configurado para devolver esse cabeçalho. Como podemos descobrir essa configuração?

Mais DevTools: Network

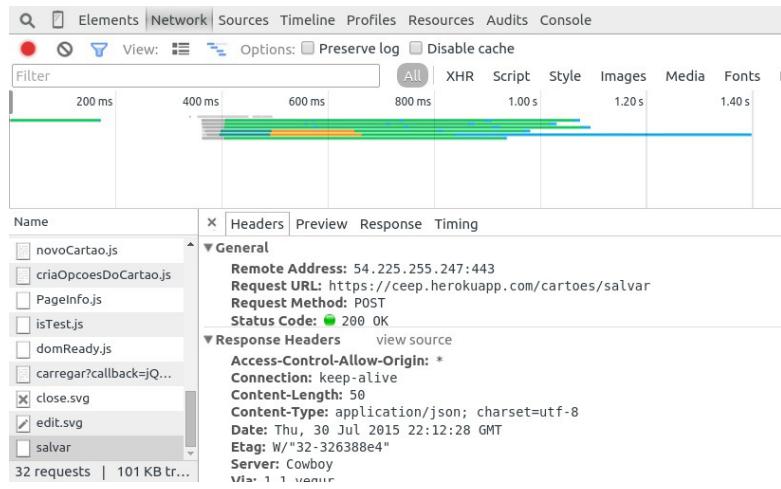
No DevTools o navegador reúne os vários dados relacionados às comunicações externas na aba *Network*.



Na coluna **name** temos as urls de todas as requisições feitas, e nas próximas colunas várias informações a respeito de cada uma delas:

- Method: o método da requisição (GET ou POST)
- Status: código de resposta do servidor
- Initiator: linha de código que iniciou a requisição
- Size: quantidade de dados baixados
- Time: tempo total desde a chamada até ela ser completada
- Timeline: gráfico mais detalhado do tempo gasto

Podemos ainda saber mais detalhes clicando no nome da requisição. A aba **Headers** contém as informações que não são relevantes para o usuário. São em sua maioria protocolos de comunicação da web. Umas dessas informações é exatamente o cabeçalho do CORS:



5.5 SAME ORIGIN POLICY E JSONP

O CORS é a solução correta para lidar com o problema do Same Origin Policy. O problema é que não é suportado em navegadores antigos. Uma outra solução é usar JSONP.

Para contornar a limitação e permitir que qualquer um possa fazer uso das informações fornecidas por uma aplicação, foi criado o JSONP (JSON with Padding) que, em vez de retornar um objeto do JavaScript puro, retorna uma chamada de função:

```
qualquerFuncao({
  local: "Caelum",
  horaInicial: "19:00",
  horaFinal: "23:00"
})
```

Para que as informações sejam aproveitadas então, é necessário que o nosso script tenha uma função com o mesmo nome que o servidor responde, tratando essa resposta. Essa função é o que chamamos de função de *callback*. O utilitário `ajax` do jQuery já trata automaticamente o nome da função de *callback*: basta definirmos que queremos utilizar o formato JSONP:

```
$.ajax({
  url: "http://www.servidor.com/servico",
  dataType: "jsonp",
  success: function (data, textStatus, jqXHR) {
    // Essa é a função de callback
  }
});
```

`$.getJSON` e JSONP

Quando utilizarmos a função `$.getJSON` para obter JSONP, precisamos adicionar na url um parâmetro e um valor. Geralmente esse parâmetro tem o nome "callback" e o valor pode ser qualquer nome que estipularmos. O retorno virá dentro de uma função com este nome:

```
$.getJSON("http://servidor.com.br/servico?callback=nomeFuncao",
  function(retorno) {
```


CAPÍTULO 6

MELHORANDO NOSSO APP COM BOAS PRÁTICAS DE CÓDIGO

6.1 O PROBLEMA DOS ESCOPOS EM JAVASCRIPT

Vários dos códigos que escrevemos até agora no projeto criavam variáveis soltas. Por exemplo, quando fizemos o código do JSON que precisava do email do usuário, criamos uma variável `usuario` e logo a chamada ao JSON:

```
var usuario = "seu.email@example.com.br";  
$.getJSON(...);
```

O problema desse código é que ele a variável é **global**. Isso quer dizer que qualquer parte da aplicação, qualquer arquivo .js, qualquer função, agora tem acesso a essa variável `usuario`. Os problemas são muitos.

A qualquer hora o valor dela pode ser alterado. Há possibilidade de conflito, afinal `usuario` é um nome bastante comum que pode acabar sendo usado em outra parte da aplicação. No fim, temos um código mais confuso de ler, menos elegante e mais propenso a erro.

Para corrigir, só há uma solução: esconder a variável e tirá-la do escopo global.

Escopo de função

A linguagem JS só tem dois escopos possíveis: o global que vimos e o escopo de função. Ou a variável é acessível por todos, ou ela é uma variável local de função.

Portanto, para resolver o problema que citamos do global, a única solução é colocar numa função:

```
function executa() {  
    var usuario = "seu.email@example.com.br";  
}
```

Nesse momento, sabemos que a variável `usuario` não é mais legível fora da função. Por isso, precisamos colocar a chamada ao JSON lá dentro também, para poder ter acesso ao valor de `usuario`:

```
function executa() {  
    var usuario = "seu.email@example.com.br";  
    $.getJSON(...);  
}
```

Claro que só criar a função não é suficiente. Afinal ela precisa ser chamada com `executa();` para rodar o código. Isso deixa escancarado outro problema: o nome da função é o nosso novo global.

Na tentativa de esconder a variável `usuario` dentro da função, acabamos criando uma função global com nome `executa`. O mesmo problema volta. A chance de alguém reescrever esse nome em outro lugar. O código feio e difícil de manter.

6.2 IIFE: IMMEDIATELY INVOKED FUNCTION EXPRESSIONS

Funções anônimas

A solução para não ter um nome de função no escopo global? Deixá-la anônima, como já fizemos tantas vezes antes.

```
function() {
  var usuario = "seu.email@exemplo.com.br";
  $.getJSON(...);
}
```

Mas criamos um novo problema: como chamar uma função anônima?

IIFE

Em JavaScript, tudo é um valor, uma expressão. Inclusive funções. Podemos atribuir funções a variáveis e podemos passar funções como parâmetro, como já fizemos tantas vezes.

Sabendo disso, é possível pensar num truque. Definir a função sem nome não nos permite chamá-la depois (ela não tem nome!). Mas a própria definição da função em si devolver um valor funcional que pode ser invocado imediatamente. Parece difícil mas é invocar a função logo que ela é definida:

```
(function() {
  var usuario = "seu.email@exemplo.com.br";
  $.getJSON(...);
})();
```

Repare o `()` no final. Isso indica invocação. Mas ao invés de passar o nome da função antes, passamos a própria função. Uma expressão que devolve a função como valor.

O nome disso é *Expressões de Funções Imediatamente Invocáveis*. Ou IIFE. É uma excelente prática para encapsulamento e organização de código.

6.3 ORGANIZAÇÃO DE ARQUIVOS JAVASCRIPT

Uma prática bastante importante para códigos mais organizados é de ter pequenos arquivos, pequenas partes da aplicação que se juntam no todo. Arquivos pequenos são mais fáceis de ler e de manter. Diminuem a chance de duas pessoas do time mexerem ao mesmo tempo. E deixam tudo mais

organizado e encapsulado.

Mais ainda. Agora que sabemos IIFE, podemos pensar que cada arquivo pode ser encarado como uma funcionalidade independente e que não deve vazar coisas no escopo global desnecessariamente.

Ou seja, podemos quebrar o código que fizemos antes em pequenos arquivos totalmente encapsulados em IIFEs individuais e autocontidas.

Se pensarmos nas funcionalidades atuais do projeto e em como isolá-las, talvez chegaremos em algo assim:

- sincronizacao.js
- mudaLayout.js
- novoCartao.js
- ajuda.js
- adicionaCartao.js

Depois, na página, vamos importar cada parte separadamente.

E performance?

Há uma preocupação, bastante válida, com relação a performance. A boa prática diz que menos arquivos, menos requests, são melhores para performance.

Isso nem sempre é verdade, mas em geral é uma boa diminuir sim, em especial se usarmos o HTTP/1.1 clássico.

Mas em desenvolvimento queremos quebrar tudo isso para um melhor código, mais organizado. Parecem coisas conflitantes, mas não são. Mais pra frente, estudaremos ferramentas que permitem fazer mudanças como essa na nossa aplicação automaticamente, antes de colocar o código em produção. Teremos vários arquivos isolados em desenvolvimento e um arquivo juntado apenas em produção. Por enquanto, nosso foco é na organização do código.

6.4 MÓDULOS EM JAVASCRIPT

As IIFE encapsulam totalmente o código dentro delas. Isso é ótimo para encapsulamento. Mas e quando precisamos acessar algo de fora? Quando queremos uma IIFE para encapsular tudo mas precisamos expor alguma coisa para as pessoas chamarem em outras partes do programa?

É o que faremos com uma sintaxe de Módulos em JavaScript.

É só uma função

Lembre que uma IIFE é uma função. Nós conseguimos o encapsulamento apenas pensando no

escopo local da função que isola as variáveis. O que mais temos em funções? Retorno.

Se pensar bem, uma função é totalmente isolada do mundo exterior (encapsulamento) mas se comunica com o mundo externo através de parâmetros (entrada) e do retorno (saída). Toda função pode retornar alguma coisa para quem chama. Mesmo nossas IIFEs.

Um novo comportamento

Lembra do nosso módulo de usuário? Ele encapsula a variável e o comportamento do AJAX associado ao usuário.

```
(function(){
  var usuario = "seu.email@exemplo.com.br";

  $.getJSON(...);
})()
```

Imagine que queremos uma função que *descarta o usuário atual*. Uma função simples, que apenas zera a variável `usuario`. Como essa variável faz parte da nossa IIFE, está encapsulada, a nova função precisa estar lá dentro também:

```
(function(){
  var usuario = "seu.email@exemplo.com.br";

  $.getJSON(...);

  function descartaUsuario() {
    usuario = undefined;
  }
})()
```

Uma coisa a se reparar é temos uma função dentro da outra. Sem problemas. No JavaScript funções são objetos normais e podem aparecer em qualquer lugar.

A nova função `descartaUsuario` faz exatamente o que queremos. Mas ela está dentro da IIFE. Isso quer dizer que só lá dentro vamos enxergá-la. Apenas código da IIFE pode chamar a função nova.

E se quisermos chamar essa função em outro ponto na aplicação?

Retornando algo útil

Nossa IIFE pode retornar alguma coisa. Uma variável, uma string, uma função, qualquer coisa. E quem chama a IIFE pode receber esse retorno numa variável.

E se retornarmos a `descartaUsuario` ?

```
var descartaUsuario = (function(){
  var usuario = "seu.email@exemplo.com.br";

  $.getJSON(...);
```

```
function descartaUsuario() {
    usuario = undefined;
}

return descartaUsuario;
})();
```

Repare no retorno da função. E repare na primeira linha, onde pegamos esse retorno e guardamos numa variável. Essa variável (por acaso com mesmo nome, mas não precisaria) está no escopo global. Isso quer dizer que qualquer parte do programa pode chamá-la.

IIFEs com retorno são um bom jeito de encapsular um código e só expor aquilo que realmente é necessário.

Anônimos

É possível simplificar ainda mais e deixar a função interna sem nome e retorná-la diretamente:

```
var descartaUsuario = (function(){
    var usuario = "seu.email@example.com.br";

    $.getJSON(...);

    return function() {
        usuario = undefined;
    };
})();
```

6.5 MÓDULO COM OBJETOS

Nosso módulo parece bom. Mas e se quisermos expor mais de uma função para o exterior? Como é tudo uma função, sabemos que só é possível retornar uma única coisa. Então como retornar uma estrutura que permita várias funções? Objetos JavaScript.

Muitas funções

O módulo de usuário estava assim, com apenas uma função:

```
var descartaUsuario = (function(){
    var usuario = "seu.email@example.com.br";

    $.getJSON(...);

    function descartaUsuario() {
        usuario = undefined;
    }

    return descartaUsuario;
})();
```

Imagine que queremos mais comportamentos. Além de `descartaUsuario`, agora queremos também um `atualizaDados` que faz uma chamada JSON para pegar dados mais atuais do usuário.

Como retornar as duas coisas? E como receber as duas?

```
var ??? = (function(){
    var usuario = "seu.email@exemplo.com.br";

    $.getJSON(...);

    function descartaUsuario() {
        usuario = undefined;
    }

    function atualizaDados() {
        // chama JSON de usuário
    }

    return ???;
})();
```

Agrupando funções num objeto

Ao invés de devolver uma única função, podemos devolver um simples objeto JavaScript que agrupe as duas funções, nomeando-as. Quem recebe o módulo, recebe agora um objeto completo com vários comportamentos pendurados.

```
var moduloUsuario = (function(){
    var usuario = "seu.email@exemplo.com.br";

    $.getJSON(...);

    function descartaUsuario() {
        usuario = undefined;
    }

    function atualizaDados() {
        // chama JSON de usuário
    }

    return {
        descarta: descartaUsuario,
        atualizaDados: atualizaDados
    };
})();
```

Quem quiser chamar esse código, pode fazer:

```
moduloUsuario.atualizaDados();
```

Anônimos

É possível, claro, configurar o objeto com as funções direto no momento do retorno com funções anônimas:

```
var moduloUsuario = (function(){
    var usuario = "seu.email@exemplo.com.br";

    $.getJSON(...);

    return {

```

```

descarta: function () {
    usuario = undefined;
},
atualizaDados: function () {
    // chama JSON de usuário
}
};

})();

```

6.6 DEPENDÊNCIAS COM IIFE

Agora que nosso módulo está completo, conseguimos disponibilizar para fora a função `adicionaCartao`. Mas a aplicação ainda pode dar o mesmo erro : **adicionaCartao is not defined**. O que pode acontecer para o javascript falar pra gente que não existe uma função chamada `adicionaCartao`?

Para poder usar essa função, o navegador deve ler a função primeiro. E com tantos arquivos, podemos facilmente esquecer de importar o `controladorDeCartoes.js` no HTML. Ou mais, podemos simplesmente errar a ordem das tags `<script>` ;

Para funcionar, a IIFE do `novoCartao.js` **depende** de um objeto externo, o objeto `controladorDeCartao`. Como a IIFE é uma função, toda dependência externa pode ser passada como parâmetro, basta criar uma variável interna para receber esse valor e passar normalmente nos parênteses que invocam a função:

```

(function(controlador){

    $(".novoCartao").submit(function(event){

        var campoConteudo = $(".novoCartao-conteudo");

        var conteudo = campoConteudo.val()
                        .trim()
                        .replace(/\n/g, "<br>");

        if(conteudo){
            controlador.adicionaCartao(conteudo);
        }

        campoConteudo.val("");

        event.preventDefault();

    });
})(controladorDeCartoes);

```

Essa estratégia nos dá duas vantagens. A primeira é que podemos dar o nome que quisermos para a variável internamente, potencialmente deixando nosso código menos verboso. A segunda é que no final da IIFE temos uma lista de dependências que ela precisa para funcionar. Assim, basta ler a última linha para saber quais módulos precisam ser importados no HTML antes.

6.7 USE STRICT

Vimos que o Javascript permite muita liberdade na hora de escrever nosso código. Muitas vezes, ao invés de dar algum erro, o javascript simplesmente tenta executar o que ele entendeu, mesmo que não seja o que queremos. Mas isso pode atrapalhar o nosso desenvolvimento, se escrevemos alguma coisa errada ele não vai nos dizer. Por isso, o ECMAScript 5 introduziu o modo estrito no JavaScript. Sua intenção é permitir que desenvolvedores escolham por uma "versão" do JavaScript na qual alguns dos erros mais comuns são tratados de maneira diferente.

Todos os navegadores modernos suportam o strict mode. Para ter uma lista detalhada, você pode consultar <http://caniuse.com/use-strict>.

As principais proibições deste modo são:

- declaração `with`
- omissão de `var` na declaração de variáveis
- objetos com propriedades duplicadas
- funções com parâmetros duplicados

Este modo pode ser ativado incluindo-se o texto "use strict" globalmente ou localmente dentro de cada função. A segunda opção é mais recomendada, uma vez que o modo estrito declarado globalmente pode quebrar código já existente:

```
(function(){
  "use strict";

  // código omitido
})();
```

O PODER DOS EVENTOS

7.1 ACOPLAMENTO DE CÓDIGO

Temos uma tarefa bastante complicada na aplicação que é sincronizar os dados locais dos cartões com nosso servidor remoto via AJAX. Temos um botão que o usuário pode clicar para disparar a sincronização. Mas dessa forma, o usuário pode esquecer de sincronizar os dados, e perder todas as informações. Podemos resolver esse problema fazendo ele sincronizar automaticamente toda vez que adicionamos um novo cartão.

Quando mais precisamos fazer a sincronização automática? Remover, editar... qualquer alteração que o usuário fizer é interessante já salvar no servidor para ele não perder a mudança.

O que deve acontecer quando uma sincronização precisa ser feita? No momento já temos duas lógicas independentes: o AJAX e mudança no estilo do botão. Mas podemos colocar mais, como executar um log de operações ou desabilitar o botão de sincronizar. Ou alguma outra coisa.

Ou seja, temos vários tipos de lógica que podem precisar ser disparados nesse momento de sincronizar, além do AJAX propriamente dito.

Entendendo o problema do acoplamento

É importante compreender o cenário que estamos discutindo aqui. Uma certa coisa pode ser disparada de pontos diferentes da aplicação. E diferentes ações podem ser executadas por causa disso.

Pensando em código, podemos pensar numa função simples. Imagine que criamos uma função `sincroniza()`:

```
function sincroniza() {
    // faz o AJAX pra sincronizar com o servidor
}
```

Aí temos vários pontos da aplicação que vão disparar essa sincronização. Por exemplo o botão de sincronizar e o botão que adiciona novos cartões:

```
$("#sync").click(function(){
    // chama a funcao de sincronizacao
    sincroniza();
});

$(".novoCartao").submit(function(){
    // ... resto da logica
});
```

```
// chama a função de sincronização
sincroniza();
});
```

Ou seja, 2 lugares na aplicação chamando a função. E podemos ter mais.

Mas não é só isso. Queremos também que, além da sincronização, seja mostrado um spinner para usuário, que o formulário de adição seja desabilitado e que um log seja feito. Onde adicionar esses comportamentos?

Se formos colocar em quem chama a sincronização, vamos acabar com código espalhado:

```
$("#sync").click(function(){
    mostraSpinner();
    desabilitaFormulario();
    logDeSincronizacao();
    sincroniza();
});

$(".novoCartao").submit(function(){
    // ... resto da lógica

    mostraSpinner();
    desabilitaFormulario();
    logDeSincronizacao();
    sincroniza();
});
```

Repare que já tivemos o trabalho de colocar cada lógica numa função separada, mas ainda temos que chamar todas elas em todos os lugares. Não é uma boa ideia e vamos esquecer alguma.

Outra opção seria manter apenas a chamada ao `sincroniza()` e colocar toda as outras lógicas (spinner, formulário, log) dentro dela. Mas aí teremos uma grande função super complicada cheia de comportamento. Falamos que uma função com muita responsabilidade, pouco coesa.

Dando um passo atrás, em JavaScript, temos outros cenários com essa mesma limitação que já foram resolvidos. Quando clico em um botão qualquer, várias ações podem ser executadas, sem que uma precise conhecer (acoplar) a outra. E vários elementos podem disparar cliques em pontos diferentes da aplicação.

O que faz tudo isso funcionar muito bem no mundo do JS? **Eventos**.

7.2 EVENTOS CUSTOMIZADOS

Eventos naturalmente apresentam um modelo de organização de código desacoplado. Há um acontecimento qualquer (um clique por exemplo) que é nosso evento. Pessoas interessadas nesse acontecimento podem registrar um listener (callback) e ser avisada quando ele aconteceu.

O ponto é que podemos ter inúmeros listeners para o mesmo evento, de forma independente. Um não conhece o outro, o que gera código desacoplado. E podemos ter vários pontos na aplicação que

disparam esse mesmo evento. Pense num submit, que pode ser disparado ao clicar no botão ou dar enter no formulário; tanto faz para os listeners quem e como disparou o evento.

Tudo isso de eventos é interessante e nativo nos navegadores. Usamos click, submit, mouseover etc. Mas podemos fazer coisas ainda mais poderosas, com interações não previstas pelo JS, com **Eventos customizados**.

Criando eventos customizados com a função `trigger`

Podemos criar nossos próprios eventos, algo que não necessariamente representa uma interação do usuário, mas sim uma interação entre as ações do sistema. No nosso caso, queremos avisar todo o sistema que **precisamos sincronizar**. Com a ajuda do jQuery, podemos facilmente usar eventos customizados. O evento nem precisa ser criado, propriamente dito, basta escrever o comando que dispara um evento passando o nome dele.

```
$(document).trigger("precisaSincronizar");
```

Quando esse comando é executado, um evento é disparado em todo o documento. É como clicar em um botão, ou passar o mouse por cima, mas qualquer elemento na página pode estar esperando esse evento acontecer. Agora falta registrar alguém que ficará escutando pelo evento. Para isso, usamos a função `on` como estamos acostumados, com a diferença de que trabalharemos com o evento que criamos:

```
$(document).on("precisaSincronizar", function(event) {
    // posso fazer o ajax
});

$(document).on("precisaSincronizar", function(event) {
    // e eu posso fazer o spinner
});
```

No exemplo acima, para o evento "precisaSincronizar", temos dois 'observadores' que serão executados quando o evento for disparado. Conseguimos acesso à informação que foi pendurada no momento do disparo do evento por meio do segundo argumento da função de callback.

Podemos adicionar quantos 'observadores' quisermos para este evento, desacoplando-os da fonte geradora, facilitando, assim, a manutenção de nosso código.

A FUNÇÃO ONE

Muitas vezes queremos processar um evento apenas uma vez. É por isso que existe a função **one**. Ela assemelha-se à função `on` com a diferença de que parará de escutar o evento após ele ter processado uma primeira vez. Exemplo:

```
$(".botao").one("click", function(event) {  
    alert("Executarei este alerta apenas uma vez");  
});
```

Como desacoplar nosso exemplo

A refatoração para usar eventos sincronizados no nosso exemplo tem duas etapas. Primeiro, os locais que causam sincronização agora vão simplesmente dar um trigger no evento:

```
$("#sync").click(function(){  
    $(document).trigger("precisaSincronizar");  
});  
  
$(".novoCartao").submit(function(){  
    // ... resto da logica  
  
    $(document).trigger("precisaSincronizar");  
});
```

Agora, temos 4 funções que criamos com as lógicas independentes que devem ser disparadas no momento da sincronização. As 4 podem ser listeners desse evento customizado:

```
$(document).on("precisaSincronizar", mostraSpinner);  
$(document).on("precisaSincronizar", desabilitaFormulario);  
$(document).on("precisaSincronizar", logDeSincronizacao);  
$(document).on("precisaSincronizar", sincroniza);
```

Repare como nenhuma função conhece outra função. Tudo desacoplado. Tudo com eventos. E usando um evento criado por nós. Eventos customizados são uma ótima forma de melhorar o design do nosso código.

7.3 CONTENTEDITABLE

Agora que a função de adicionar e remover cartões está pronta, vamos criar uma nova funcionalidade: a edição de cartões. Para editar cartões, precisamos que o usuário passe pra gente o novo conteúdo. O HTML possui várias tags para entrada de dados: `input`, `textarea`. Uma vez criado, precisamos de uma forma para o usuário salvar a edição. Naturalmente surge a idéia de usar um `<form>`:

```
<form class="edita">  
    <textarea class="edita-conteudo"></textarea>  
    <button>Salvar</button>  
</form>
```

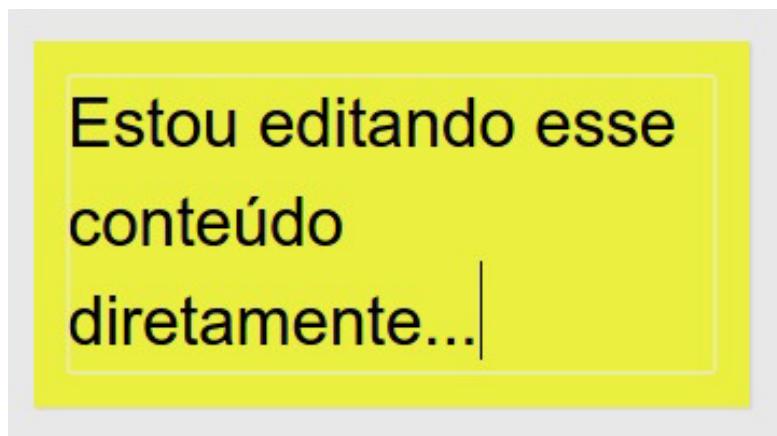
Note que ele é muito parecido com o formulário de adicionar. Não podemos reaproveitá-lo? O HTML é o mesmo, mas o que deve acontecer quando salvamos muda drasticamente quando queremos editar ou só criar um novo cartão. Além disso, quando estamos editando um cartão no fim da página, o foco teria que mudar para o começo. Como fica a usabilidade da aplicação?

O ideal é podermos editar diretamente o valor do conteúdo do cartão. No HTML5 é possível também transformar qualquer elemento HTML em editável. Basta colocar o atributo `contenteditable`:

```
<p class="cartao-conteudo" contenteditable>Texto que usuario pode mexer</p>
```

Agora o usuário pode clicar no parágrafo, digitar texto, apagar. Realmente editar seu conteúdo.

Depois, em JavaScript, podemos pegar o valor digitado apenas olhando a propriedade `textContent` desse elemento.



7.4 ELEMENTOS INTERATIVOS E O FOCO

Alguns elementos têm o poder de interagir com o usuário. Exemplos são as tags `<input>`, `<textarea>`, `<a>` e qualquer elemento com o atributo `contenteditable`.

Tais elementos ganham a capacidade de ser acessíveis não só com o mouse, mas também com o teclado. Isso quer dizer que tanto com o mouse, quanto através de navegação com o teclado, o usuário consegue chegar ao elemento e interagir com ele. Dizemos que o elemento é **focável**.

Foco no javascript

Quando podemos fazer a sincronização da edição do conteúdo do cartão?

Sempre que um elemento é focado, um evento `focus` é disparado. Assim, é possível executar um código sempre que o elemento for focado.

```
$(".elemento").on("focus", function(){
```

```
});
```

Da mesma forma, existe um evento disparado quando o foco sai do elemento. O evento `blur`. Podemos usar desses eventos para fazer a sincronização.

```
$(".elemento").on("blur", function(){
    $(document).trigger("precisaSincronizar");
});
```

Durante alguma interação do usuário poder ser necessário que algum elemento arbitrário ganhe ou perca foco. No javascript é possível fazer isso com as funções `focus` e `blur`, respectivamente.

```
//elemento ganha foco
elemento.focus();

//elemento perde foco
elemento.blur();
```

Acessibilidade

É recomendado que qualquer elemento que seja focado tenha uma resposta visual. É possível estilizar um elemento focado com pseudo classe `focus`, no css.

```
.elemento:focus {
    border: 2px solid blue;
}
```

Alguns estilos são aplicados por padrão a elementos que estão focados. No Google Chrome elementos focados ficam com o que parece ser uma borda azul. No Firefox o elemento focado fica com uma borda preta e tracejada.

Elementos no caminho do foco

Nem todos os elementos da página podem ser focados. Assim, alguns elementos não são acessíveis via navegação por teclado. Se o conteúdo daquele elemento for importante para a compreensão da navegação, é possível incluí-lo no caminho do TAB, ou seja, dar a capacidade para que um certo elemento seja focado. Basta utilizar o **tabindex**.

Tabindex

É possível adicionar um atributo tabindex a qualquer elemento do HTML. O atributo pode ter 3 tipos de valores.

Um tabindex maior que 0 indica a posição do elemento dentro da navegação. O valor 1 indica que ele é o primeiro a ser focado. Seguido dos elementos com valor 2, depois os com valor 3... O elemento a seguir é o primeiro a ser focado na navegação da página

```
<p tabindex="1">
    ...
</p>
```

Um `tabindex` igual a 0 diz que o elemento é acessível via navegação com o teclado e que pode ser focado. Porém, o elemento não tem uma posição específica dentro da navegação. Ele segue a hierarquia do DOM.

```
<p tabindex="0">  
  ...  
</p>
```

Um `tabindex` menor que 0 diz que o elemento não deve ser focado via navegação de teclado, porém, deve ganhar a habilidade de ser focado caso seja clicado.

```
<p tabindex="-1">  
  ...  
</p>
```

7.5 EVENTOS E PERFORMANCE DO SITE

Muitos eventos no JavaScript são disparados numa frequência bem alta. Por exemplo, quando o usuário faz scroll na página, o `onscroll` é chamado sucessivamente. Quando o mouse se move, o `onmousemove` é chamado centenas de vezes conforme o cursor move. E assim por diante.

Se adicionamos um listener em algum evento desses, corremos o risco de executar nosso callback muitas vezes. E se a função fizer algo potencialmente pesado, podemos ter problemas de performance.

Evitando chamadas em excesso

O jeito é evitar que a função de callback seja chamada para cada disparo do evento. De alguma forma retardar o evento. Há várias formas de lidar com isso. Um padrão bem conhecido é o "debounce".

Ele é muito útil quando estamos interessados apenas no final da interação do usuário. Por exemplo: o usuário faz scroll mas quero executar algo apenas quando ele parar de fazer scroll. Ou o usuário está digitando algo e quero executar algo ao fim da digitação.

Ou seja, queremos executar nosso listener quando certo evento acabar de ser chamado.

O Debounce Pattern

A ideia é executar um certo listener apenas se o evento não tiver sido chamado há algum tempo. Esperamos um tempo determinado (1 segundo por exemplo) para ver se o evento não vai ser disparado novamente. Se não for, é um sinal que ele parou de ser chamado então podemos invocar nosso callback.

Como implementar isso? É mais simples do que parece.

Vamos por partes. Com o `setTimeout` conseguimos agendar nossa função para ser executada no futuro. Podemos fazer então algo assim:

```
// evento que dispara muito  
window.onscroll = function() {
```

```
setTimeout(function(){
    // lógica pesada
}, 1000);
}
```

Esse código adiciona um listener de scroll que, ao invés de executar a lógica complicada dentro dele, adia 1000ms (1s) sua execução. Repare que se o scroll disparar 1000 vezes ainda faremos 1000 vezes a lógica complicada, apenas adiamos 1s.

O que queremos é dar o próximo passo: evitar que a lógica complicada seja chamada 1000 vezes. Ela deve ser chamada apenas uma vez 1s após o término do scroll. O segredo é usar o `clearTimeout` para impedir que a execução anterior seja realmente chamada. O código é curto mas exige uma certa abstração para se entender que a lógica vai ser chamada somente uma vez no fim:

```
var timer;

// evento que dispara muito
window.onscroll = function() {
    clearTimeout(timer);

    timer = setTimeout(function(){
        // lógica pesada
    }, 1000);
}
```

Repare que guardamos o último timer na variável global `timer`. Antes de agendar o próximo timeout, removemos o anterior. Isso efetivamente deixa apenas um timer ativo por vez. E é o último que foi chamado.

Efetivamente, o que fazemos é zerar o contador a cada vez que o evento é disparado. Assim, a lógica só será executada depois que o usuário parou de fazer aquela interação repetida.

THROTTLE PATTERN

Outra forma de lidar com lógicas pesadas em eventos que são disparados diversas vezes seguidas é simplesmente reduzir proporcionalmente a quantidade de vezes que a lógica vai ser chamada. Chamamos isso de Throttle Pattern.

Podemos fazer isso facilmente usando um contador:

```
var contador = 1;
window.onscroll = function(){
    if(contador >= 10){
        //lógica pesada
        contador = 0;
    }
    contador++;
}
```

No exemplo acima, nossa lógica pesada será executada uma vez para cada dez disparos do evento de scroll

Generalizando o Debounce

Se precisarmos de debounce em muitos lugares da aplicação, não vamos querer replicar esse tipo de código. Podemos definir uma função `debounce` que recebe qualquer função de lógica e transforma numa versão com debounce.

A ideia é chamar assim:

```
window.onscroll = debounce(function(){
    // lógica pesada
});
```

Essa função `debounce` é uma generalização da ideia anterior devolvendo uma função para ser usada no callback:

```
function debounce(callback) {
    var timer;
    return function() {
        clearTimeout(timer);
        timer = setTimeout(callback, 1000);
    };
}
```

7.6 DELEGAÇÃO DE EVENTOS

Dado o código abaixo, pede-se que quando qualquer `` for clicado, o elemento com `class=list`a tenha a cor de fundo alterada.

```
<ul class="lista">
```

```

<li class="lista-trocaCor" data-cor="#000">Cor 1</li>
<li class="lista-trocaCor" data-cor="#BBB">Cor 2</li>
<li class="lista-trocaCor" data-cor="#542">Cor 3</li>
<li class="lista-trocaCor" data-cor="#EAE">Cor 4</li>
</ul>

```

Será que é necessário colocar um event listener em cada opção de cor? Quantos event listeners teremos se nossa página tiver 20 caixas?

Uma alta quantidade de event listeners pode causar *problemas de performance* no seu site. Para diminuir o número de event listeners necessários, podemos utilizar algumas técnicas.

Event Bubbling

Quando clicamos na página, como o navegador sabe qual é o elemento alvo do click? Na verdade é muito simples: cada elemento é uma "caixa" na página, então o navegador olha qual caixa contém o ponto que foi clicado. Portanto se clicarmos em uma ``, o navegador vê que o ponto do clique está dentro da "caixa" que é o ``. Mas essa é a única caixa que contém esse ponto?

Como o `` está dentro de um ``, quando clicamos em qualquer `` dessa lista, podemos dizer que estamos clicando também, na ``. Assim, no caso abaixo, a mensagem será exibida se clicarmos tanto nos ``'s quanto na ``.

```

$(".lista").click(function(){
  console.log("Lista foi clicada");
});

```

No seguinte caso, qual mensagem aparece primeiro se um algum `` da lista for clicado?

```

$(".lista").click(function(){
  console.log("Lista foi clicada");
});

$(".lista-trocaCor").click(function(){
  console.log("Item foi clicado");
});

```

Na verdade estamos clicando **nos dois** elementos (e também em todos os pais na hierarquia). A primeira mensagem será: "Item foi clicado". Os eventos são disparados primeiramente no elemento alvo do evento, ou seja, o `` e depois seguem na ordem de parentesco do HTML.

Damos a esse comportamento dos eventos o nome de **bubbling**.

Delegação de eventos

Para reduzir o número de event listeners na troca de cores utilizarmos o comportamento de **bubbling** dos eventos para fazer o que chamamos de **delegação de eventos**.

Queremos chamar um lógica sempre que alguém clica numa cor. Como todas as cores são `` dentro da ``, vamos colocar o **listener** na ``.

```
$(".lista").click(function(){
    //código que troca a cor
});
```

Para garantir que usuário clicou em algum item e não em algum outro ponto da lista precisamos saber quem realmente foi clicado. Para isso, trabalharemos com o objeto `event` que o browser disponibiliza em todo evento. O objeto `event` tem um atributo chamado `target` que representa o elemento alvo da interação. No caso, o clique.

```
//é preciso pedir o objeto com as informações do evento
$(".lista").click(function(event){
    //elemento que foi clicado
    event.target
});
```

Para garantir que o elemento clicado é um item da lista, basta acrescentar uma verificação:

```
$(".lista").click(function(event){
    if(event.target.classList.contains("lista-trocaCor")){
        var itemDaLista = event.target
    }
});
```

Podemos facilmente mudar a cor da `` agora:

```
$(".lista").click(function(event){
    if(event.target.classList.contains("lista-trocaCor")){
        var itemDaLista = event.target;
        $(this).css('background-color', itemDaLista.attr("data-cor"));
    }
});
```

CAPÍTULO 8

EXERCÍCIO: REMOVENDO CARTÕES COM JAVASCRIPT

8.1 OBJETIVO

1. Com JavaScript podemos fazer várias coisas. Uma das principais é alterar a página em resposta às ações do usuário. Nossa primeira funcionalidade será remover o cartão quando clicarmos no botão de remover:

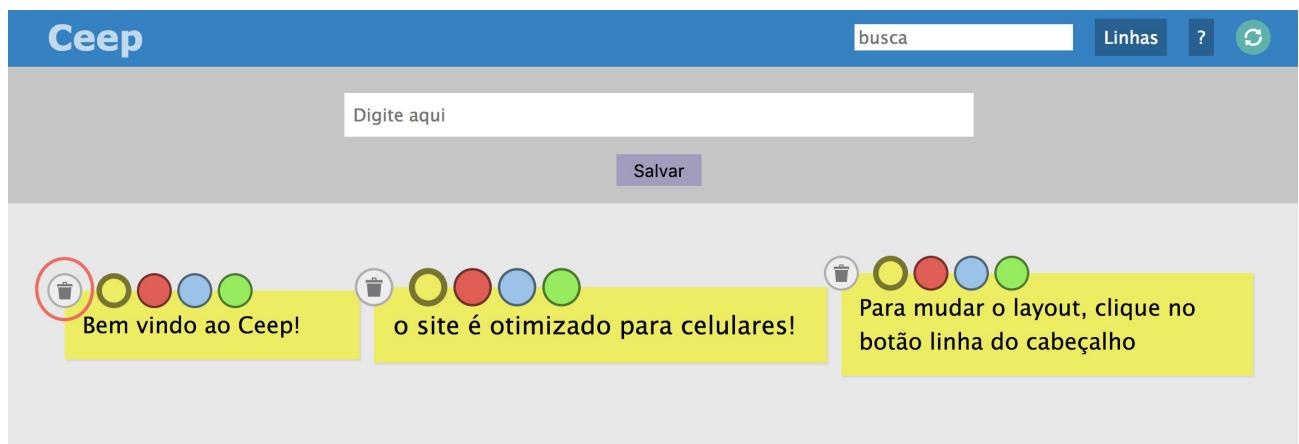


Figura 8.1: Botão remover

Clicando no botão do primeiro cartão, ele deve ser removido e a página ficará assim:

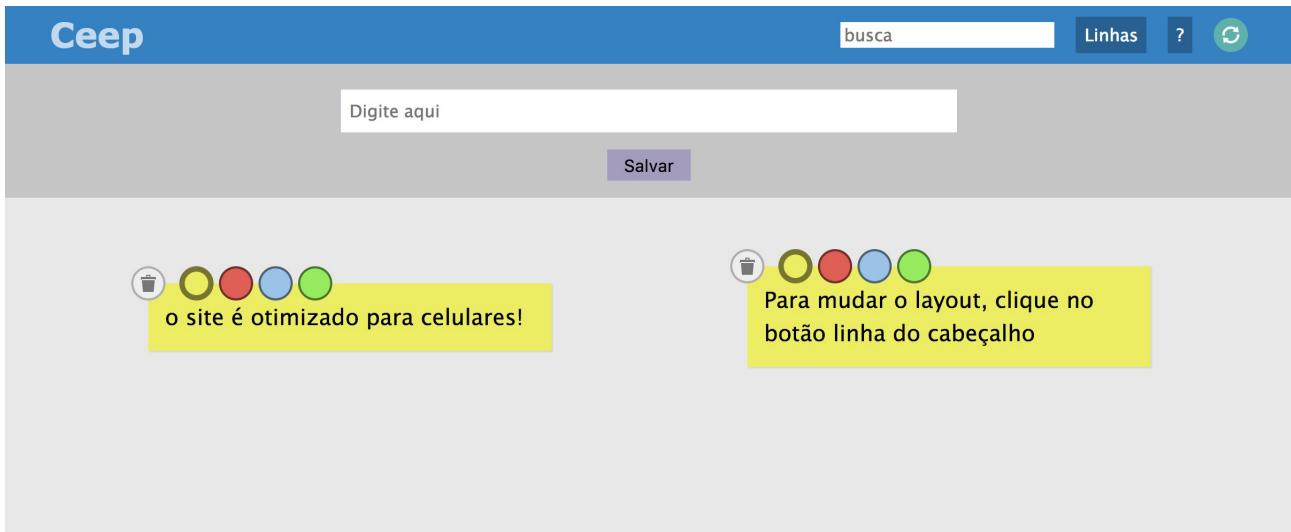


Figura 8.2: Primeiro cartão foi removido

Obs. é sempre bom tentar fazer o código sem olhar o passo a passo da apostila. Use o passo a passo da apostila para validar o que você entendeu e o que você está com dúvida.

8.2 PASSO A PASSO COM CÓDIGO

1. Para removermos um elemento, precisamos saber em que momento ele será removido. Nesse caso, quando o usuário clicar no botão. O código de remoção será executado no evento de `click` do botão.

```
# index.html
```

```
<button class="opcoesDoCartao-remove opcoesDoCartao-opcao" tabindex="0" onclick="this.parentNode.parentNode.remove()">
```

2. Agora que conseguimos remover o primeiro cartão, podemos replicar o código anterior para todos os outros cartões.

3. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 9

EXERCÍCIO: MUDANDO O CONTEÚDO DO BOTÃO QUANDO ELE É CLICADO

9.1 OBJETIVO

1. Usuário irá clicar no botão que está com o texto **Linhas**:

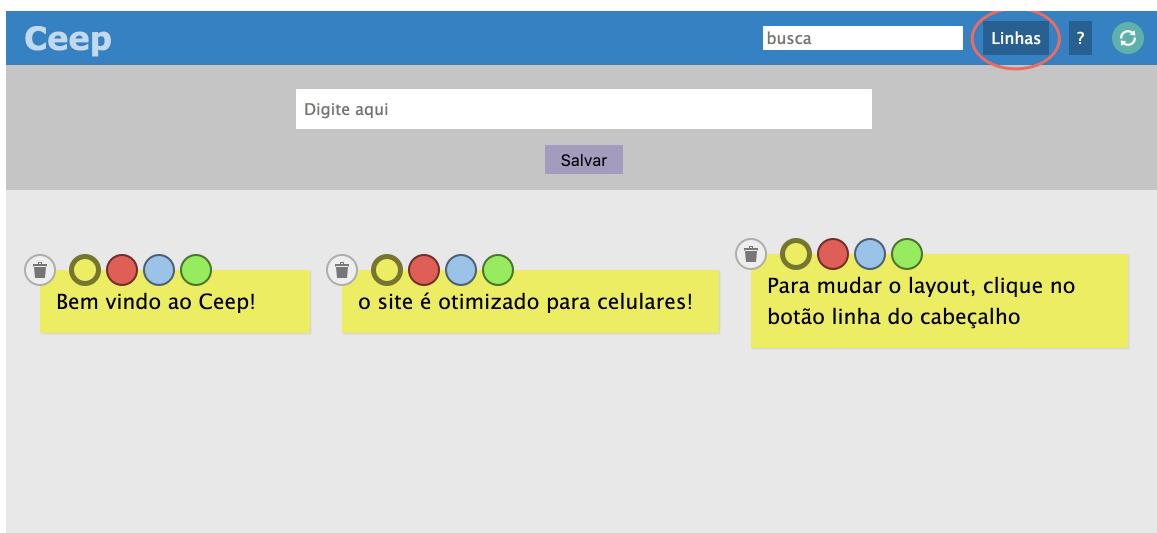


Figura 9.1: Estado da tela antes do usuário clicar no botão Linhas

2. Assim que esse clique acontecer nós teremos que mudar o texto do botão para **Blocos**:

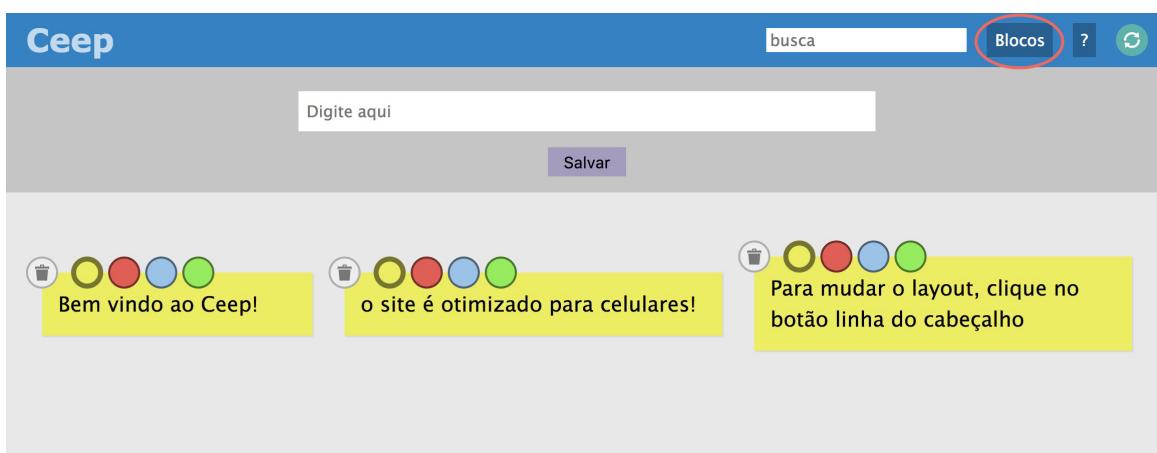


Figura 9.2: Estado da tela após o usuário ter clicado no botão Linhas

Obs. é sempre bom tentar fazer o código sozinho para validar o que você entendeu e o que você está com dúvida.

9.2 PASSO A PASSO COM CÓDIGO

1. Precisamos falar para o browser que queremos executar uma mudança na tela quando o usuário clicar no botão **Linhas**. Esse código é mais complexo do que o que tínhamos nos botões de remover. Por isso, faremos ele em um arquivo JavaScript separado. Crie uma pasta com o nome de **js** e dentro dela uma outra pasta com nome **opcoesDaPagina**. Dentro dessa pasta crie o nosso primeiro arquivo JavaScript do curso que terá o nome de **btnMudaLayout.js**. Para carregá-lo junto com a página, inclua a tag script no html antes do fechamento da tag body:

```
# index.html

<script src="js/opcoesDaPagina(btnMudaLayout.js)"></script>
```

2. Dentro do arquivo **btnMudaLayout.js** teremos que criar a nossa função **mudaTexto** e dizer que ela será executada no evento de **click** do botão.

```
# js/opcoesDaPagina/btnMudaLayout.js

const btn = document.querySelector('#btnMudaLayout')

function mudaTexto() {
    if(btn.textContent == 'Blocos') {
        btn.textContent = 'Linhas'
    } else {
        btn.textContent = 'Blocos'
    }
}

btn.onclick = mudaTexto
```

3. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 10

EXERCÍCIO: MUDANDO O LAYOUT DOS CARTÕES NO MURAL

10.1 OBJETIVO

1. Usuário irá clicar no botão que está com o texto **Linhas**:

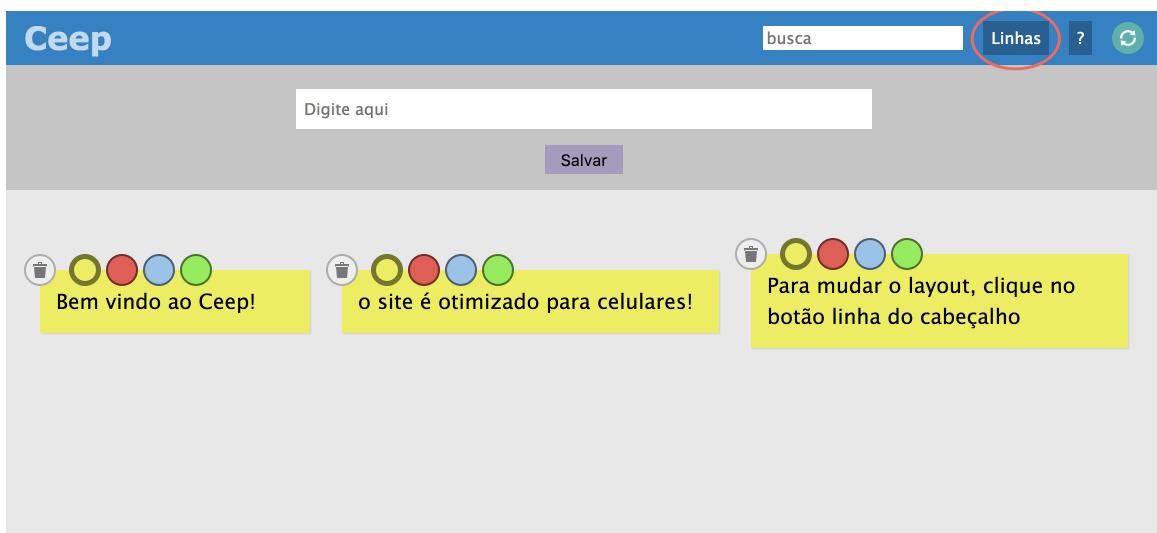


Figura 10.1: Estado da tela antes do usuário clicar no botão Linhas

2. Assim que esse clique acontecer nós teremos que mudar a disposição dos cartões que estão dentro da `<section>` que tem a `class="mural"`. Os cartões que estavam em linha (um do lado do outro) agora ficam como blocos (um embaixo do outro). Estamos mudando o estilo do mural quando o botão é clicado. Temos que chegar no seguinte resultado:

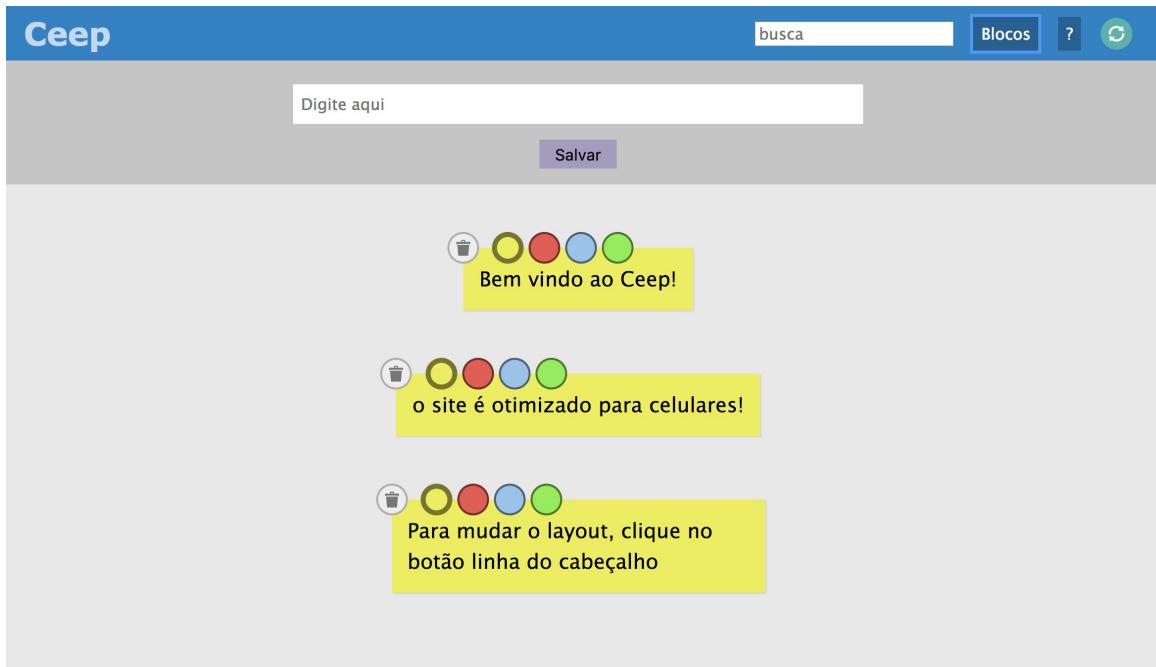


Figura 10.2: Estado da tela após o usuário ter clicado no botão Linhas

Obs. é sempre bom tentar fazer o código sozinho para validar o que você entendeu e o que você está com dúvida.

10.2 PASSO A PASSO COM CÓDIGO

1. Se formos até o browser e abrirmos o nosso `index.html` e clicarmos no botão **Linhas**, veremos que o texto do botão já é alterado, mas não a disposição dos cartões no mural. Para mudar a disposição, precisamos alterar o estilo da tag `<section>` que tem a `class="mural"`, adicionando uma classe `mural--linha`. Mas só podemos fazer isso quando o evento `click` for disparado no botão. Temos duas coisas acontecendo quando o botão é clicado: o texto do botão é alterado e o estilo do mural é alterado. Por isso, não podemos mais usar o `onclick`, já que só poderíamos ter um Event Listener para o evento.

```
# js/opcoesDaPagina/btnMudaLayout.js
```

Remova o `onclick`. No lugar do `onclick` colocaremos um `addEventListener`

```
btn.onclick = mudaTexto
btn.addEventListener('click', mudaTexto)
```

Agora podemos adicionar a função `mudaLayout` para mudar a classe do mural quando o botão é clicado.

```
const mural = document.querySelector('.mural')
function mudaLayout() {
  mural.classList.toggle('mural--linha')
}
```

```
btn.addEventListener('click', mudaLayout)
```

2. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

EXERCÍCIO: FUNCIONALIDADES COM PROGRESSIVE ENHANCEMENT

11.1 OBJETIVO

1. Algumas funcionalidades ainda não estão prontas mas mesmo assim mostramos os botões delas. Por exemplo, o botão da sincronização. Se não temos uma determinada funcionalidade não deveríamos exibir o html na tela do usuário final.

Como só implementamos a funcionalidade de mudar o layout do mural, nossa página deve exibir apenas o botão com **id** `#btnMudaLayout` .

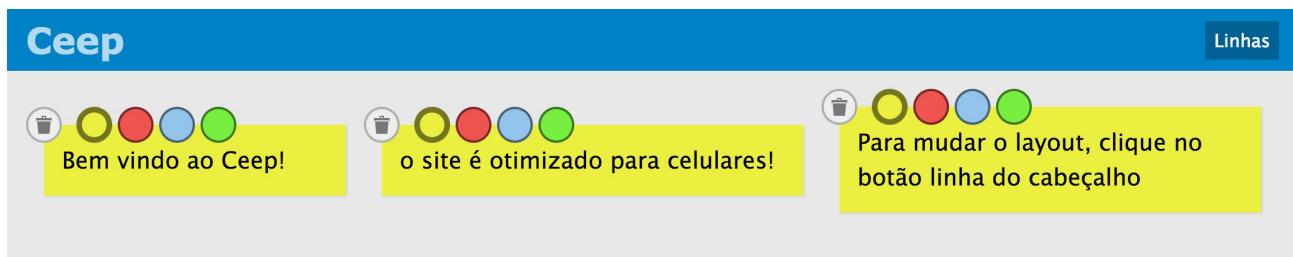


Figura 11.1: Mostrando apenas o botão muda layout

Obs: Como os cartões serão criados pelo usuário mais à frente, por meio do JavaScript, não nos importaremos com as funcionalidades dentro deles no momento.

11.2 PASSO A PASSO COM CÓDIGO

1. Não podemos sair removendo os htmls da página, pois, sem html não teremos nem o que mostrar depois. Por isso, vamos escondê-los apenas visualmente, com CSS. Para isso crie a classe **no-js** no CSS que terá o código que esconde elementos.

```
# css/reset.css
```

Coloque esse código no final do arquivo, depois de todo o código que já temos:

```
.no-js {
  display: none !important;
}
```

2. Agora que temos a class **no-js** no CSS, devemos colocar essa mesma classe nos elementos que

queremos esconder da nossa página.

```
# index.html
```

Campo de busca:

```
<input type="search" placeholder="busca" id="busca" class="opcoesDaPagina-opcao no-js">
```

Botão muda layout:

```
<button id="btnMudaLayout" class="opcoesDaPagina-opcao opcoesDaPagina-botao no-js">
```

Botão de ajudas:

```
<button id="btnAjuda" class="opcoesDaPagina-opcao opcoesDaPagina-botao no-js">
```

Botão da sincronização:

```
<button id="btnSync" class="opcoesDaPagina-opcao opcoesDaPagina-botao botaoSync botaoSync--sincronizado no-js">
```

Formulário para criar cartão:

```
<form class="formNovoCartao container no-js">
```

3. Agora que os elementos já estão escondidos. Precisamos mostrar o botão muda layout, pois a gente já implementou essa funcionalidade. Para isso, vamos remover a classe **no-js** quando o código da funcionalidade for executado:

```
# js/opcoesDaPagina/btnMudaLayout.js
```

Coloque esse código no final do arquivo, depois de todo o código que já temos:

```
btn.classList.remove('no-js')
```

4. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 12

EXERCÍCIO: ANIMANDO A REMOÇÃO DO CARTÃO

12.1 OBJETIVO

1. Nossa CEEP está começando a ganhar vida com suas funcionalidades, porém quando removemos um cartão ele some de uma forma meio bruta. Para deixar isso mais suave, vamos fazer uma transição de opacidade via CSS no momento em que o cartão for removido. Para isso, precisamos adicionar a classe `cartao--some` no cartão quando clicarmos no botão remover.

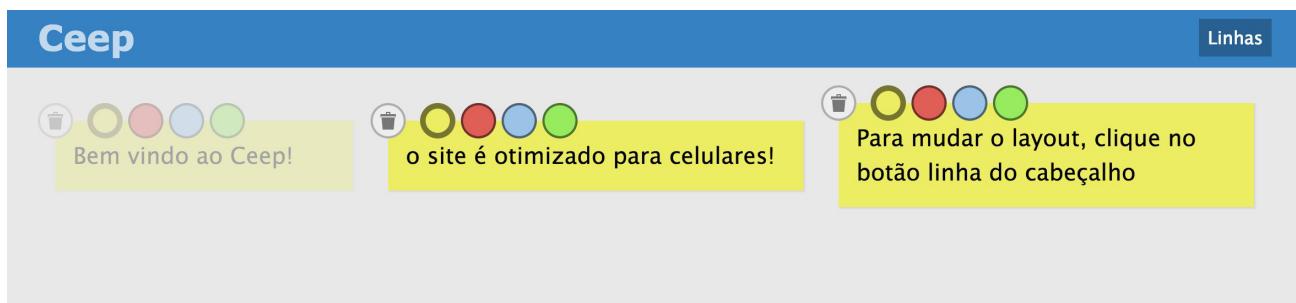


Figura 12.1: Cartão some aos poucos antes de sumir completamente

12.2 PASSO A PASSO COM CÓDIGO

1. A funcionalidade de remover o cartão agora ficou mais complexa e difícil de manter dentro do atributo `onclick` no HTML. Por isso vamos remover todos os `onclick` dos botões de remover e depois criar os códigos em um arquivo JavaScript separado.

```
# index.html
```

Remova o atributo `onclick`, mas mantenha o botão:

```
<button class="opcoesDoCartao-remove opcoesDoCartao-opcao" tabindex="0" onclick="this.parentNode.parentNode.remove()>
```

Obs: Lembre-se que temos três cartões, e portanto, 3 botões com `onclick` para remover.

2. Agora precisamos adicionar o código novo. Vamos escrever o código em um novo arquivo. Seguindo nosso padrão, criaremos a pasta `opcoesDoCartao` e dentro dela um arquivo chamado `remove.js`. Para carregá-lo junto com a página, inclua a tag `script` no html antes do fechamento da tag

body:

```
# index.html

<script src="js/opcoesDoCartao/remove.js"></script>
```

3. No código da remoção, adicionaremos um **Event Listener** do evento de **click**. Quando o evento for disparado, adicionaremos a classe `cartao--some` no cartão, o que iniciará a transição. Após a transição acabar, podemos remover o cartão do mural chamando a função `cartao.remove()`. Desta vez usaremos apenas funções anônimas como handlers dos Event Listeners.

```
# js/opcoesDoCartao/remove.js
```

```
// Seleciona Elemento
const btn = document.querySelector('.opcoesDoCartao-remove')
// Adiciona um Evento de click sem usar onclick e criando função anônima
btn.addEventListener('click', function() {
    const cartao = btn.parentNode.parentNode
    cartao.classList.add("cartao--some")
    cartao.addEventListener("transitionend", function(){
        cartao.remove()
    })
})
```

4. Neste ponto do código algo meio triste acontece. A variável `btn` já foi criada anteriormente, e por isso o console do navegador nos mostra o seguinte erro: **Uncaught SyntaxError: Identifier 'btn' has already been declared**.

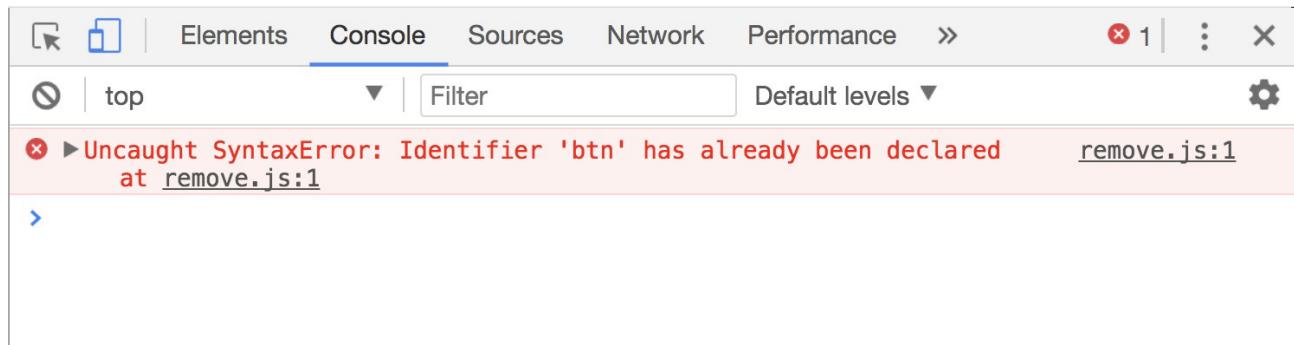


Figura 12.2: Variável `btn` já foi declarada

Para solucionar o erro, criaremos um novo escopo utilizando uma IIFE em volta de todo nosso código do exercício anterior.

```
# js/opcoesDoCartao/remove.js
```

Na primeira linha do código abra a IIFE:

```
; (function(){
```

Na última linha do código feche a IIFE:

```
}())
```

5. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 13

EXERCÍCIO: ADICIONANDO EVENTO DE REMOVER PARA TODOS OS CARTÕES

13.1 OBJETIVO

1. Nesse momento, os cartões podem ser removidos com transição na opacidade, porém o único que pode ser removido no momento é o primeiro cartão. O evento deve de alguma forma ser adicionado para todos os botões que removem cartão.

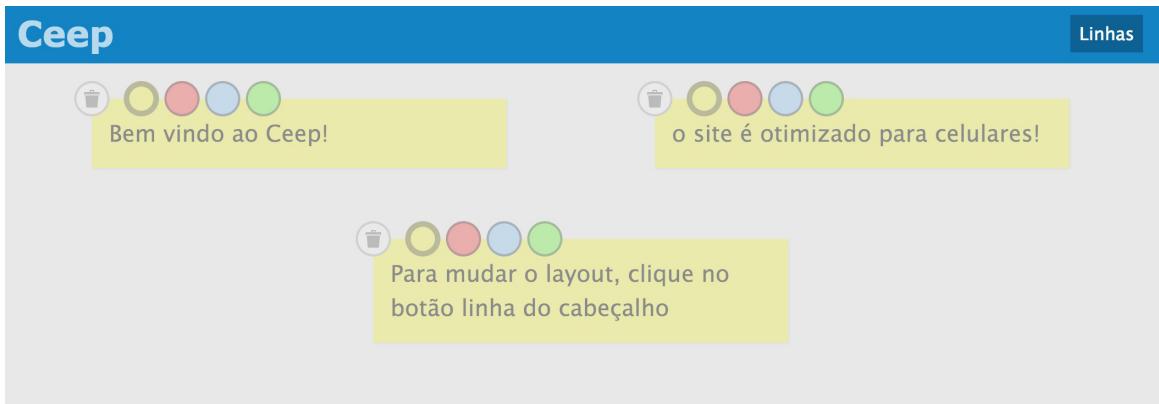


Figura 13.1: Cartão some aos poucos antes de sumir completamente

13.2 PASSO A PASSO COM CÓDIGO

1. Agora estamos trabalhando com todos os botões, ao invés de mexer com só um elemento vamos pegar uma lista com todos eles.

```
# js/opcoesDoCartao/remove.js
```

Vamos alterar o nome da variável para deixar claro que agora temos vários botões.

```
const btn = document.querySelector('.opcoesDoCartao-remove')
const btns = document.querySelectorAll('.opcoesDoCartao-remove')
```

2. Se tentarmos adicionar o Event Listener direto na variável `btns` o seguinte erro irá aparecer:

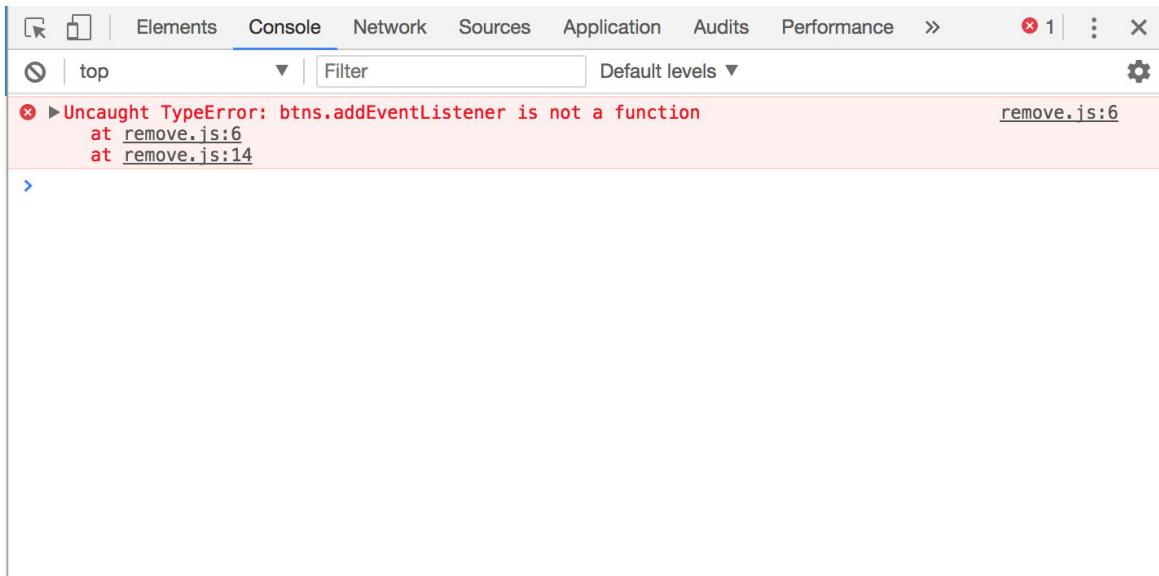


Figura 13.2: Erro quando o Event Listener é aplicado direto na variável btns

A variável `btns` representa uma lista de elementos e não um só. Não podemos usar `addEventListener` numa lista de elementos, apenas em elementos individuais. Por isso, temos que percorrer todos os buttons que temos nessa lista para que cada elemento individualmente receba o evento de click da remoção.

Ao final seu código estará assim:

```
# js/opcoesDoCartao/remove.js

;(function(){
    const btns = document.querySelectorAll('.opcoesDoCartao-remove')
    for(let i = 0; i < btns.length; i++) {
        btns[i].addEventListener('click', function() {
            const cartao = btns[i].parentNode.parentNode
            cartao.classList.add("cartao--some")
            cartao.addEventListener("transitionend", function(){
                cartao.remove()
            })
        })
    }
})()
```

3. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 14

EXERCÍCIO: MOSTRAR OPÇÕES SOMENTE DO CARTÃO QUE ESTAMOS MEXENDO

14.1 OBJETIVO

- As opções dos cartões estão aparecendo o tempo todo e poluem nossa página. Queremos que elas só apareçam quando formos mexer no cartão.

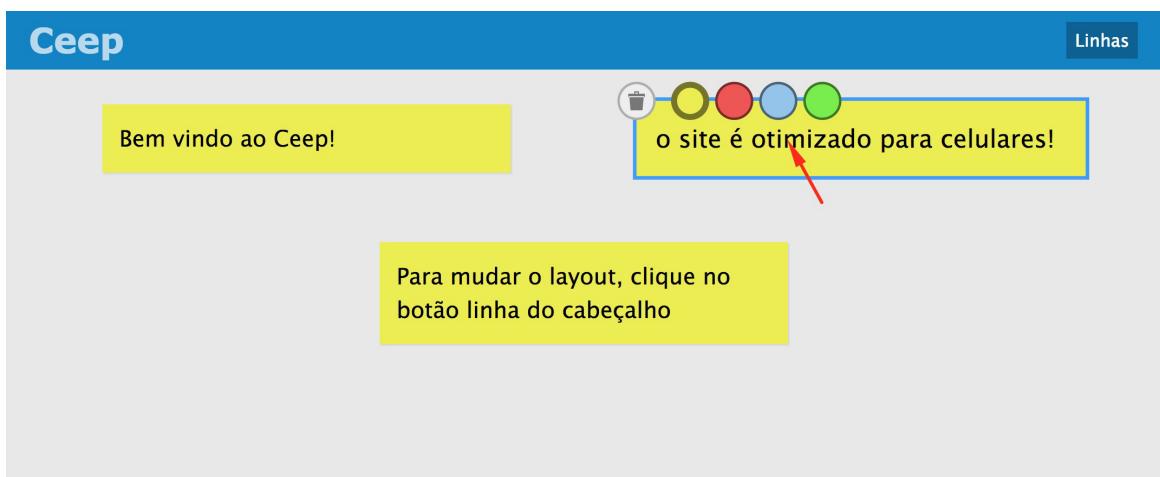


Figura 14.1: As opções aparecem somente no hover e no focus do cartão

14.2 PASSO A PASSO COM CÓDIGO

- Os usuários que acessarem o Ceep por meio de mouse ou trackpad vão conseguir acessar os elementos via `hover` então esse será nosso primeiro passo no código. Por padrão os elementos sempre aparecem, vamos inverter isso e liberar a visibilidade das opções do cartão somente no `hover`. Ao invés de usar `display:none` para fazer os elementos sumirem, vamos fazer um mix de `pointer-events` com `opacity` deixando assim mais fácil fazermos alguma animação com `transition`.

```
# css/opcoesDoCartao.css
```

Após a última linha do arquivo, adicione o código abaixo.

```
.opcoesDoCartao-opcao {  
  pointer-events: none;  
  opacity: 0;  
  transition: opacity .3s;  
}
```

```
.cartao:hover .opcoesDoCartao-opcao {  
    opacity: 1;  
    pointer-events: auto;  
}
```

2. Vale lembrar que nem sempre os usuários irão ter teclado e mouse, quem navegar no site via tab (deficientes visuais por exemplo) ou via dispositivos mobile não irão conseguir fazer um hover nos cartões para ver as opções. Para resolver esse caso, precisamos permitir que os **cartões** tenham acesso via tab, que acontece por meio do `focus`, e para liberar isso nos elementos precisamos colocar o atributo do html `tabindex="0"` neles.

```
# index.html
```

Primeiro cartão:

```
<article id="cartao_1" class="cartao">  
<article id="cartao_1" tabindex="0" class="cartao">
```

Segundo cartão:

```
<article id="cartao_2" class="cartao">  
<article id="cartao_2" tabindex="0" class="cartao">
```

Terceiro cartão:

```
<article id="cartao_3" class="cartao">  
<article id="cartao_3" tabindex="0" class="cartao">
```

3. Nesse ponto, poderíamos voltar no código css que adicionamos anteriormente no arquivo **css/opcoesDoCartao.css** e adicionar junto ao seletor de hover `.cartao:hover .opcoesDoCartao-opcao` o seletor de focus `.cartao:focus .opcoesDoCartao-opcao`, só que ainda assim, quando o foco sair do `.cartao` e ir para algum dos botões, as nossas opções irão desaparecer. Precisamos fazer com que sempre que o cartão estiver **focado** ou **focarmos em algum elemento interno dele**, ele mantenha a visibilidade das opções. Como não podemos estender a funcionalidade do focus do CSS para fazer isso vamos colocar uma classe no cartão sempre que o usuário focar no cartão ou em algum elemento dentro dele. Essa classe fará com que as opções apareçam, igual o que fizemos no hover do cartão. Criaremos essa classe no CSS e adicionaremos ela no cartão lá no JavaScript, já que dependemos da interação do usuário.

```
# css/opcoesDoCartao.css
```

Iremos alterar nosso seletor que antes continha somente o hover, e vamos adicionar um seletor para quando a classe `cartao--focado` estiver ativa no cartão:

```
.cartao:hover .opcoesDoCartao-opcao {  
.cartao:hover .opcoesDoCartao-opcao,  
.cartao--focado .opcoesDoCartao-opcao {
```

4. Agora, vamos criar o código que adiciona a classe `cartao--focado` quando o usuário foca no cartão ou em algum elemento dentro do cartão. Esse código trata de uma funcionalidade do cartão,

então criaremos o arquivo **js/cartao.js** e incluiremos ele antes do fechamento da tag `<body>` :

```
# index.html
```

```
<script src="js/cartao.js"></script>
```

5. Colocaremos a classe `cartao--focado` sempre que o evento `focusin` for disparado em qualquer elemento do cartão. Como o evento `focusin` é um evento que sofre *Bubbling*, não precisamos de um Event Listener para cada elemento que pode ser focado dentro do cartão, podemos ter apenas um Event Listener no cartão todo. Podemos fazer a mesma coisa no `focusout`, o foco não está em mais nenhum elemento dentro do cartão ou no próprio cartão e assim, poderemos remover a classe `cartao--focado`.

Como precisamos adicionar o evento em todos os cartões, precisamos percorrer todos os cartões da página com um **for** e acessar cada cartão para adicionar o Event Listener.

```
# js/cartao.js
```

```
; (function(){
    const cartoes = document.querySelectorAll(".cartao")

    for(let j = 0; j < cartoes.length; j++){
        const cartao = cartoes[j]

        cartao.addEventListener("focusin", function(){
            cartao.classList.add("cartao--focado")
        })

        cartao.addEventListener("focusout", function(){
            cartao.classList.remove("cartao--focado")
        })
    }
})()
```

6. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 15

EXERCÍCIO: DELEGANDO QUEM VAI MUDAR A COR DOS CARTÕES.

15.1 OBJETIVO

1. Queremos nesse ponto, poder classificar qual o tipo dos nossos cartões, para fazer isso, vamos mudar as cores deles. Em outras palavras, o que deve acontecer é: ao clicarmos nas bolinhas coloridas dos cartões, a cor de fundo deve ser alterada. Em um cenário normal, adicionariíamos um evento de click para cada botão e o problema estaria resolvido. Só que se fizermos isso cada cartão terá 5 Event Listeners e em 3 cartões já teremos criado 15 Event Listeners. Como não sabemos quantos cartões existirão na página, isso pode ser perigoso, já que muitos Event Listeners podem impactar em alguns aspectos da performance do site. Por isso, tentaremos implementar a funcionalidade das cores com somente um Event Listener.

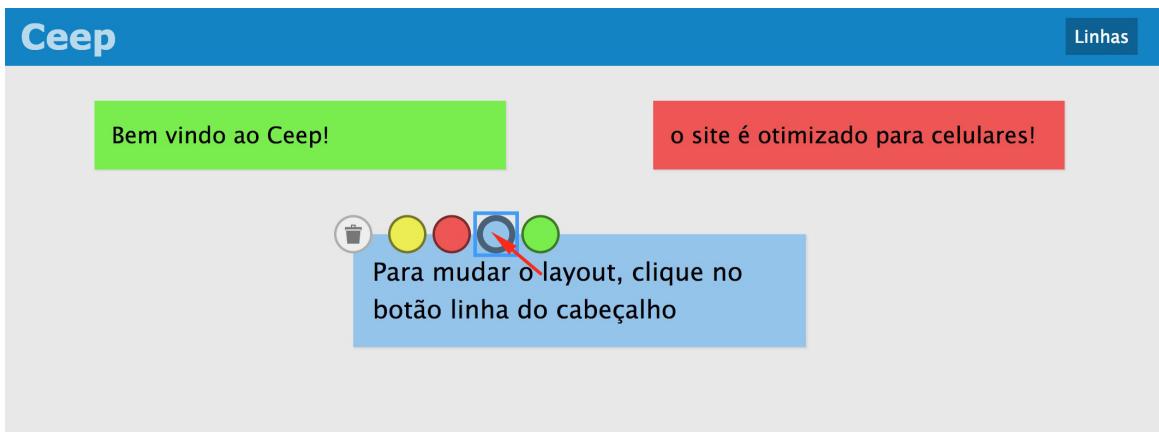


Figura 15.1: Ao clicar nas cores o cartão deve mudar a cor atual

15.2 PASSO A PASSO COM CÓDIGO

1. Os botões que mudam a cor do cartão são representados no nosso HTML por um conjunto de tags `<label>` e `<input>`. Cada botão é na verdade uma `<label>` e está sempre acompanhada de um `<input type="radio">`. Note como o atributo `for` da `<label>` é igual ao `id` do `<input type="radio">`. Dessa maneira, o `<input type="radio">` pode ficar escondido, e a única coisa que aparece na tela são as labels. Quando clicamos nos botões estamos clicando nas labels e elas selecionam o `<input type="radio">` correspondente. Sempre que temos um conjunto de opções onde apenas uma

pode ser escolhida, podemos usar esse tipo de HTML.

Clicar numa das labels e selecionar um `<input type="radio">` dispara um evento chamado **change** no input lá no JavaScript. O que vamos fazer aqui é adicionar um Event Listener nesse evento de **change** que quando for executado, pegue a cor correspondente ao botão e altere o estilo do cartão em questão.

Como não queremos um Event Listener para cada input, aproveitaremos o comportamento de Bubbling do evento **change**, e adicionaremos **um** Event Listener apenas no cartão.

```
# js/cartao.js
```

Esse código deve ser colocado antes do fechamento, dentro do **for** que percorre a lista de cartões, no arquivo **js/cartao.js**.

```
cartao.addEventListener("change", function mudaCor(event){
    const elementoSelecionado = event.target
    const isRadioTipo = elementoSelecionado.classList.contains('opcoesDoCartao-radioTipo')
    if(isRadioTipo) {
        cartao.style.backgroundColor = elementoSelecionado.value
    }
})
```

2. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

EXERCÍCIO: AJUSTANDO A NAVEGAÇÃO VIA TECLADO.

16.1 OBJETIVO

1. Já conseguimos definir o tipo de nossos cartões, porém nossa navegação via teclado não funciona perfeitamente. Um usuário que está navegando entre os botões apenas com teclado não irá clicar com o mouse, e sim, apertar **Enter** ou **Espaço** para apertar o botão. O problema é que isso não irá disparar os eventos de change dos `<input type="radio">`, pois as labels por padrão só funcionam com o click do mouse. Assim, nosso objetivo é simular um evento de `click` quando o usuário interagir com os botões das cores via teclado. No final, deve ser possível navegar e utilizar todas as opções do cartão 100% via teclado.

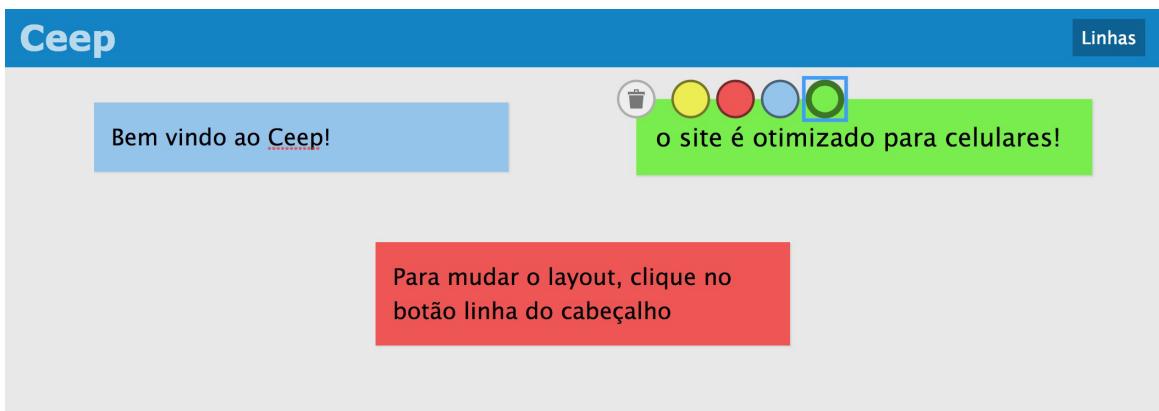


Figura 16.1: Ao apertarmos espaço ou enter durante a navegação via teclado (Tab) nas cores o cartão deve mudar a cor atual

16.2 PASSO A PASSO COM CÓDIGO

1. Quando for dado **Espaço** ou **Enter** num dos botões, um evento **keydown** é disparado neles. Para não adicionarmos um Event Listener para cada botão, adicionaremos o evento diretamente no cartão via delegate, ou seja, verificaremos se o alvo é a label que possui a classe `.opcoesDoCartao-opcao`.

```
# js/cartao.js
```

Esse código, deve ser inserido antes do fechamento do **for** que criamos no arquivo **js/cartao.js**.

```
cartao.addEventListener("keydown", function deixaClicarComEnter(event){
  if(event.target.classList.contains("opcoesDoCartao-opcao")
```

```
    && (event.key === 'Enter' || event.key === ' ')){  
        event.target.click()  
    }  
})
```

2. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

EXERCÍCIO: REMOVENDO OS CARTÕES COM DELEGATE.

17.1 OBJETIVO

1. Todas as funcionalidades do nosso cartão estão no arquivo `js/cartao.js`. Sempre que quisermos adicionar alguma funcionalidade do cartão iremos mexer nesse arquivo. Porém a primeira funcionalidade que criamos ficou isolada em um arquivo separado. Iremos trazê-la para o arquivo `js/cartao.js` e aproveitaremos para seguir o formato do delegate que aplicamos para os outros eventos.

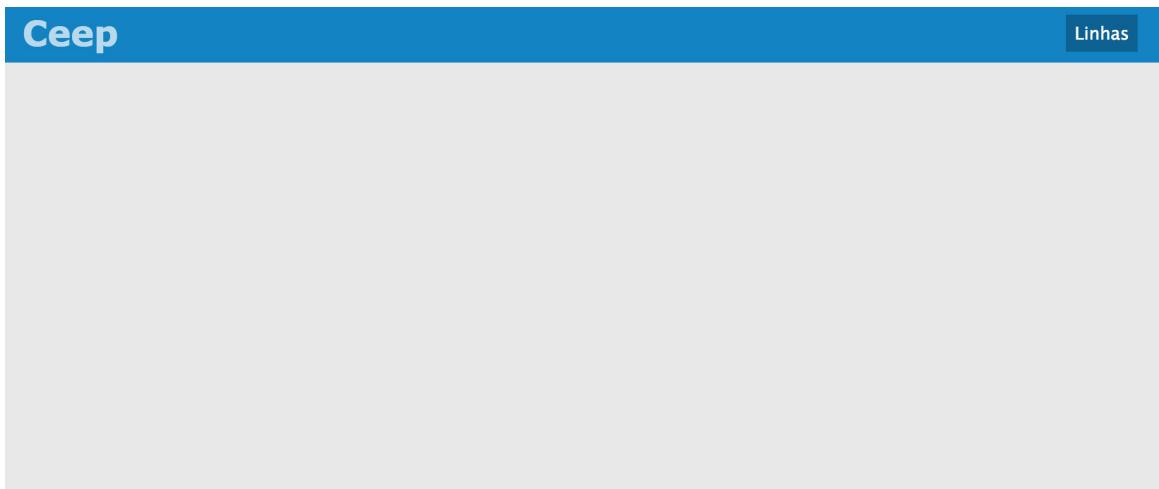


Figura 17.1: Mural com vários nada

17.2 PASSO A PASSO COM CÓDIGO

1. Precisamos alterar o código que temos no nosso arquivo `js/opcoesDoCartao/remove.js`. Pegaremos o que interessa e passaremos para o `js/cartao.js`.

- Não precisamos da IIFE já que temos no arquivo novo
- Não precisamos da variável `bt�s` e nem do `for` que acessa cada botão, já que iremos adicionar o Event Listener no cartão e não nos botões de remover.

```
# js/opcoesDoCartao/remove.js
```

```
+(function(){
```

```

const btns = document.querySelectorAll('.opcoesDoCartao-remove')

for(let i = 0; i < btns.length; i++) {
    btns[i].addEventListener('click', function() {
        const cartao = btns[i].parentNode.parentNode
        cartao.classList.add("cartao--some")
        cartao.addEventListener("transitionend", function(){
            cartao.remove()
        })
    })
}
})

```

2. Vamos pegar o código restante em **js/opcoesDoCartao/remove.js** e passar para o arquivo **js/cartao.js**, dentro do **for** que percorre todos os cartões do mural. Nossa código extraído agora deverá parar de adicionar um Event Listener em `btns[i]` e passar a adicionar o evento na variável `cartao`. Como o evento será ouvido apenas quando borbulhar no cartão, precisamos verificar se o elemento clicado é o botão remove antes de executar o código da remoção. O código ficará assim:

js/cartao.js

```

cartao.addEventListener('click', function(event) {
    const elementoSelecionado = event.target
    if(elementoSelecionado.classList.contains('opcoesDoCartao-remove')){
        cartao.classList.add("cartao--some")
        cartao.addEventListener("transitionend", function(){
            cartao.remove()
        })
    }
})

```

3. Como nosso código de remoção foi para um novo arquivo, agora podemos apagar a pasta **opcoesDoCartao** com o **remove.js**. Podemos também apagar do **index.html** a tag script referente a esse JavaScript:

index.html

```
<script src="js/opcoesDoCartao/remove.js"></script>
```

4. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 18

EXERCÍCIO: VALIDANDO ANTES DE CRIAR CARTÕES.

18.1 OBJETIVO

1. Nesse ponto, já implementamos as funcionalidades que vamos ter nos nossos cartões, já é possível editar, mudar as cores, e remover cada um deles. Porém ainda não conseguimos criar nenhum cartão novo. Nosso próximo passo é criar novos cartões, mas, antes disso, precisamos garantir que o usuário digitou algo no `<textarea>` no momento em que tentar criar o cartão (submeter o formulário). Caso o conteúdo não exista, vamos mostrar uma mensagem de erro.

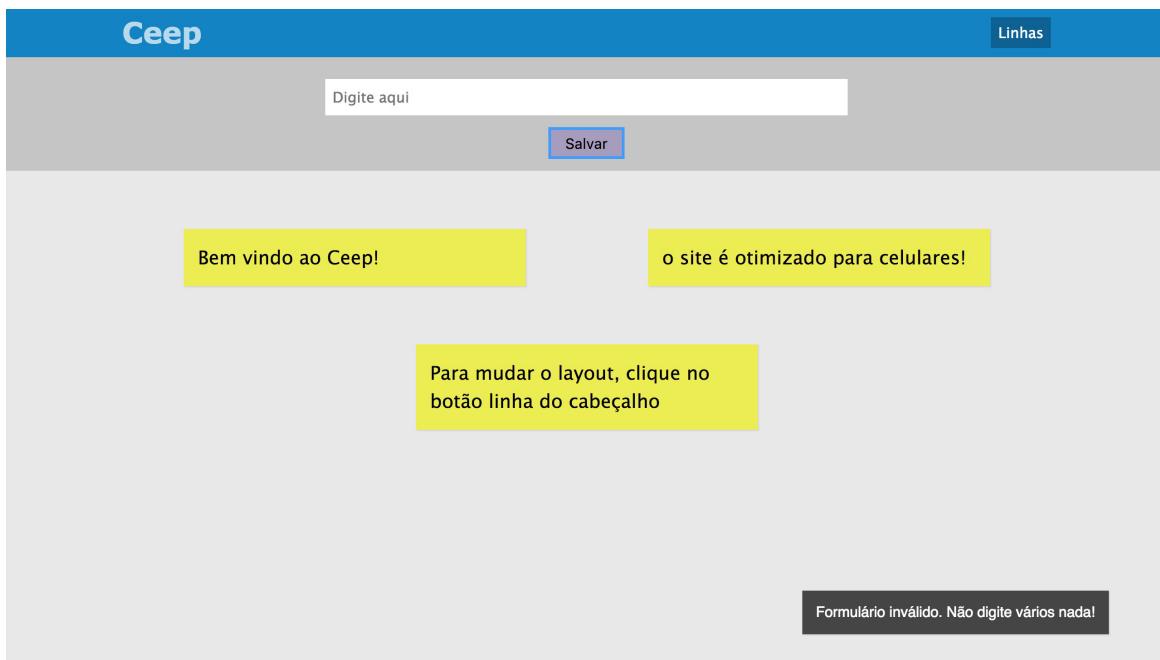


Figura 18.1: Label que informa erros de validação

18.2 PASSO A PASSO COM CÓDIGO

1. Essa nova funcionalidade vai tratar a submissão do **formNovoCartao**, assim ela vai para um arquivo novo, e seguindo nosso padrão o nome do arquivo será o mesmo dado ao componente, ou sejam, **formNovoCartao**.

```
# index.html
```

```
<script src="js/formNovoCartao.js"></script>
```

2. Com nosso arquivo criado está na hora de criar o código da funcionalidade. Faremos a validação sempre que um evento de `submit` do formulário for disparado. A função que passaremos para o Event Listener deverá inicialmente conter a instrução `event.preventDefault()`, que evita que a página seja recarregada ao fazer o submit do `<form>`. Após evitar que a página recarregue, verificaremos se o `<textarea>` está vazio ou não. Caso esteja vazio, vamos criar um pop-up para mostrar uma mensagem de erro para o usuário. Esse pop-up será uma `<div>` que contém uma class `.formNovoCartao-msg` e será inserido pela função JavaScript antes do botão submit do `<form>`.

Lembre que ao final do arquivo, precisamos remover a classe `no-js` do formulário, para que ele seja exibido.

```
# js/formNovoCartao.js

;(function(){

    const form = document.querySelector(".formNovoCartao")

    form.addEventListener("submit", function(event){
        event.preventDefault()
        const textarea = form.querySelector(".formNovoCartao-conteudo")
        const isTextAreaVazio = textarea.value.trim().length === 0
        if(isTextAreaVazio){
            const msgErro = document.createElement("div")
            msgErro.classList.add("formNovoCartao-msg")
            msgErro.textContent = "Formulário inválido. Não digite vários nada!"

            const btnSubmit = form.children[form.children.length-1]
            form.addEventListener("animationend", function(event){
                event.target.remove()
            })
            form.insertBefore(msgErro, btnSubmit)
        }
    })

    form.classList.remove("no-js")
})()
```

3. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

EXERCÍCIO: USANDO JQUERY PARA CRIAR CARTÕES DE FORMA SENSACIONAL.

19.1 OBJETIVO

1. Agora que temos nossa validação, a próxima etapa é criar os cartões quando houver conteúdo digitado no textarea.

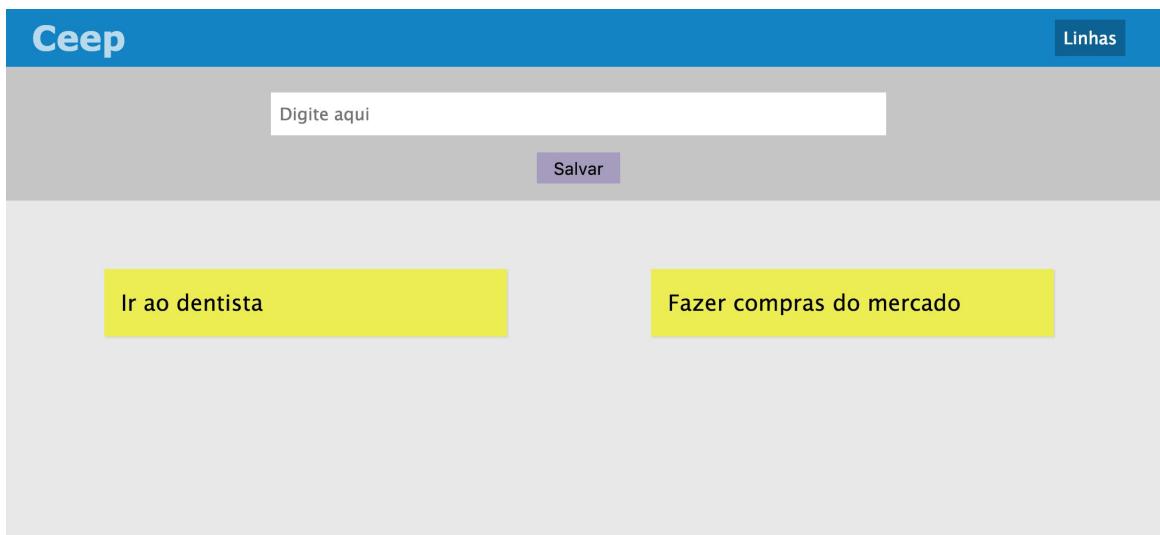


Figura 19.1: Criando cartões com conteúdo dinâmico

19.2 PASSO A PASSO COM CÓDIGO

1. O DOM possui uma API complicada para navegarmos pelos elementos da página e para conseguirmos criá-los dinamicamente. Para contornar esse ponto iremos utilizar o jQuery.

- O mesmo pode ser baixado no link <http://jquery.com/download/>
- Ou podemos pegar um link para importar ele diretamente do site da ferramenta acessando <https://code.jquery.com/>.
- Ou importar o arquivo via CDN (Content Delivery Network) <https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js>

```
# index.html
```

Insira o script com o código do jQuery antes de todas as nossas tags scripts para que o jQuery esteja disponível para todos os nossos arquivos.

```
<script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
```

2. Todo cartão possui um id que identifica o número dele. Esse id era fixo no html, mas agora que criaremos os cartões dinamicamente, precisamos contar o número de cartões para decidir qual será o id do cartão que vai ser adicionado. Quando a página carrega, temos 0 cartões, assim criaremos uma variável `numeroDoCartao` que será iniciada com o valor 0. Crie essa variável fora do Event Listener de submit do formulário, para que ela seja inicializada com 0 apenas quando a página carregar.

```
# js/formNovoCartao.js
```

```
let numeroDoCartao = 0
```

3. Agora, no Event Listener de `submit`, faremos o código que adiciona um cartão no mural. Só podemos adicionar o cartão caso o usuário tenha digitado algo no textarea. Assim, nosso código ficará dentro de um `else`, logo após o `if` da validação.

Usaremos os poderes da Template String e do jQuery para criar nosso html dinâmico de maneira declarativa, sem lidar com as complicações da API do DOM. O código html dos cartões será o mesmo que está em `index.html`. Valores dinâmicos como `id` e `conteúdo` do cartão devem ser alterados para usar os valores das variáveis no JavaScript. Após a criação do cartão, devemos inseri-lo no mural e limpar o valor do `<textarea>` que tem o conteúdo que usamos no novo cartão criado.

```
# js/formNovoCartao.js
```

```
} else {

    numeroDoCartao++
    const conteudoDoCartao = textarea.value
    const cartao = $(`

<div class="opcoesDoCartao">
            <button class="opcoesDoCartao-remove opcoesDoCartao-opcao" tabindex="0">
                <svg><use xlink:href="#iconeRemover"></use></svg>
            </button>

            <input type="radio" name="corDoCartao${numeroDoCartao}" value="#EBEF40" id="corPadrão-cartao${numeroDoCartao}" class="opcoesDoCartao-radioTipo" checked>
                <label for="corPadrão-cartao${numeroDoCartao}" class="opcoesDoCartao-tipo opcoesDoCartao-opcao" style="color: #EBEF40;" tabindex="0">
                    Padrão
                </label>

                <input type="radio" name="corDoCartao${numeroDoCartao}" value="#F05450" id="corImportante-cartao${numeroDoCartao}" class="opcoesDoCartao-radioTipo">
                    <label for="corImportante-cartao${numeroDoCartao}" class="opcoesDoCartao-tipo opcoesDoCartao-opcao" style="color: #F05450;" tabindex="0">
                        Importante
                    </label>

`)
```

```

        <input type="radio" name="corDoCartao${numeroDoCartao}" value="#92C4EC" id="corTarefa-cartao${numeroDoCartao}" class="opcoesDoCartao-radioTipo">
            <label for="corTarefa-cartao${numeroDoCartao}" class="opcoesDoCartao-tipo opcoesDoCartao-opcao" style="color: #92C4EC;" tabindex="0">
                Tarefa
            </label>

        <input type="radio" name="corDoCartao${numeroDoCartao}" value="#76EF40" id="corInspiração-cartao${numeroDoCartao}" class="opcoesDoCartao-radioTipo">
            <label for="corInspiração-cartao${numeroDoCartao}" class="opcoesDoCartao-tipo opcoesDoCartao-opcao" style="color: #76EF40;" tabindex="0">
                Inspiração
            </label>
        </div>
        <p class="cartao-conteudo" contenteditable tabindex="0">${conteudoDoCartao}</p>
    </article>
`)

$(".mural").append(cartao)

}

```

4. Como nossos cartões daqui pra frente podem ser criados pelo usuário, e nossa estrutura base para criar novos cartões já está pronta no JavaScript, não precisamos mais ter nenhum cartão no carregamento inicial da página. A tag `<section class="container mural">` pode ficar vazia.

```
# index.html

<section class="container mural" style="font-size: 1.3rem;">

</section>
```

5. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 20

EXERCÍCIO: DEVOLVENDO OS EVENTOS PARA OS CARTÕES.

20.1 OBJETIVO

1. Nossos cartões já são adicionados dinamicamente na página. Porém, eles não têm nenhum dos Event Listeners que criamos para atribuir as funcionalidades de remoção e mudança de tipo. Nosso objetivo é fazer com que os novos cartões criados possuam esses eventos, assim como os cartões estáticos que tínhamos antes.



Figura 20.1: As funcionalidades dos cartões estão de volta

20.2 PASSO A PASSO COM CÓDIGO

1. Todo o código que tínhamos dos Event Listeners estava dentro do arquivo **js/cartao.js**, onde adicionávamos eles nos cartões que estavam fixos na página. Para devolver os Event Listeners aos cartões dinâmicos, precisamos passar o código que está dentro do `for` para o arquivo **js/formNovoCartao.js**, onde os cartões dinâmicos são criados.

Note que a variável **cartao** que usávamos em **js/cartao.js** era um elemento do DOM, que acessávamos após um **document.querySelectorAll**. Agora, em **js/formNovoCartao.js**, a variável **cartao** é um elemento do jQuery, e assim, precisamos alterar algumas das funções que pertencem à API do

DOM para passar a usar a API do jQuery.

O que muda:

- .classList.add vira .addClass ;
- .classList.remove vira .removeClass ;
- .classList.contains vira .hasClass ;
- .style.propriedadeCss = 'valor da propriedade' vira .css("propriedade css", 'valor da propriedade') ;
- .addEventListener('evento', funcao) vira .on('evento', funcao) .

Outro ponto de alteração é que faremos a mudança de tipo do cartão através de uma API especial do jQuery para delegação de eventos. Note como o código da mudança de cores está menor.

js/formNovoCartao.js

Adicione os Event Listeners após a criação da variável **cartao**

```
cartao.on("focusin", function(){
    cartao.addClass("cartao--focado")
})
cartao.on("focusout", function(){
    cartao.removeClass("cartao--focado")
})

cartao.on("change", ".opcoesDoCartao-radioTipo", function mudaCor(event){
    cartao.css("background-color", event.target.value)
})

cartao.on("keydown", function deixaClicarComEnter(event){
    if(event.target.classList.contains("opcoesDoCartao-opcao") && (event.key === "Enter" || event.key === " ")){
        event.target.click()
    }
})

cartao.on('click', function(event) {
    const elementoSelecionado = event.target
    if(elementoSelecionado.classList.contains('opcoesDoCartao-remove')){
        cartao.addClass("cartao--some")
        cartao.on("transitionend", function(){
            cartao.remove()
        })
    }
})
```

2. Como todo o nosso código foi para o arquivo **js/formNovoCartao.js**, agora podemos apagar o arquivo **js/cartao.js**. Podemos também apagar do **index.html** a tag script referente a esse JavaScript:

index.html

~~_<script src="js/cartao.js"></script>~~

3. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 21

EXERCÍCIO: MOSTRANDO INSTRUÇÕES PARA OS USUÁRIOS.

21.1 OBJETIVO

1. Nossa Ceep já possui diversas funcionalidades, podemos criar cartões, alterar a visualização deles entre diversas outras coisas. Porém sempre que uma pessoa usar o Ceep, ela terá que adivinhar na raça o que ela pode fazer com a ferramenta. Para guiar a pessoa, vamos oferecer uma funcionalidade de ajuda. Ela clica no botão "Ajuda" no cabeçalho e algumas instruções aparecem na tela.

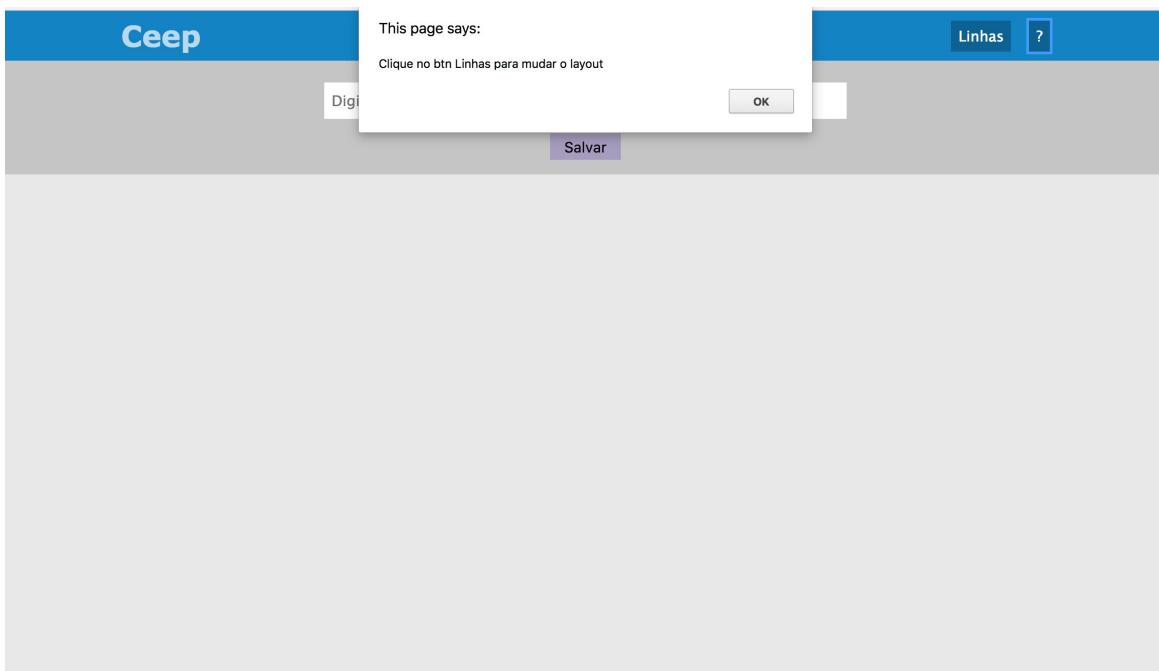


Figura 21.1: Instruções aparecendo na tela

21.2 PASSO A PASSO COM CÓDIGO

1. Como essa será uma nova funcionalidade da nossa aplicação, vamos começar criando o arquivo `js/opcoesDaPagina/btnAjuda.js` e importando ele no nosso html. O nome do arquivo e da pasta seguem o nosso padrão de componentização.

```
# index.html
```

```
<script src="js/opcoesDaPagina/btnAjuda.js"></script>
```

2. O próximo passo é criarmos o código. Quando o usuário clicar no <button> com o id btnAjuda , devemos mostrar alguns alerts com as instruções pré-definidas. Por fim, para que o botão das ajuda seja exibido, precisamos remover a class no-js.

```
# js/opcoesDaPagina(btnAjuda.js
```

```
; (function(){
    const btnAjuda = document.querySelector('#btnAjuda')
    btnAjuda.addEventListener("click", function(){
        const ajudas = [
            "Bem Vindo ao Ceep"
            , "Clique no btn Linhas para mudar o layout"
        ]

        ajudas.forEach(function(ajuda){
            alert(ajuda)
        })
    })
    btnAjuda.classList.remove('no-js')
})()
```

3. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 22

EXERCÍCIO: ALERT É HORRÍVEL. AS INSTRUÇÕES PODEM SER CARTÕES.

22.1 OBJETIVO

1. Usar o alert é uma saída para mostrar mensagens para o usuário, porém não é algo tão amigável. A principal funcionalidade do Ceep é ter um mural com informações, podemos aproveitar essa principal funcionalidade para ensinarmos o usuário, ao invés de mostrar vários alerts que bloqueiam a usabilidade da tela, vamos criar cartões que explicam as funcionalidades.

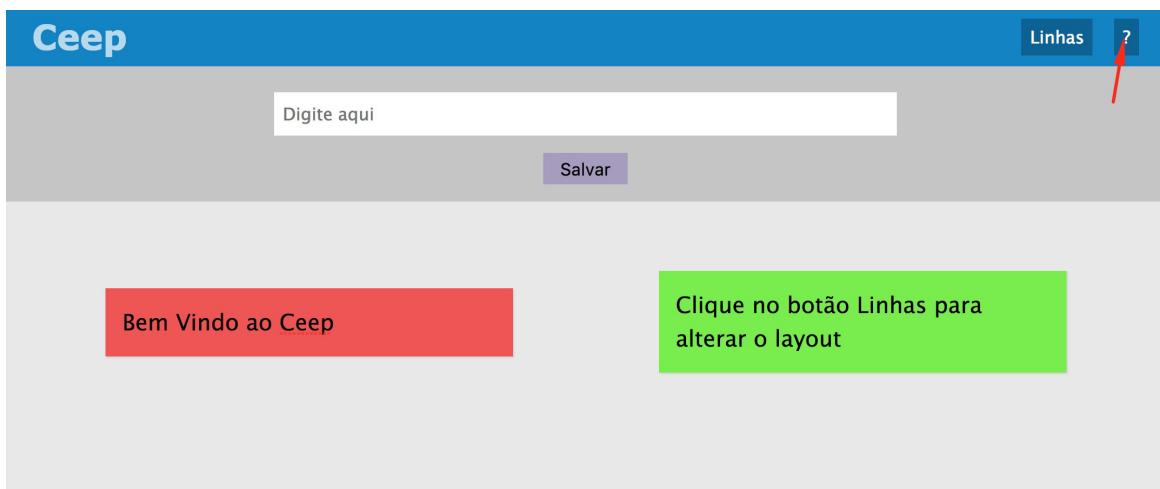


Figura 22.1: Ensinando os usuários o que é possível fazer no Ceep de forma mais amigável

22.2 PASSO A PASSO COM CÓDIGO

1. Nossas novas instruções se tornarão cartões no mural. Cada cartão de instrução terá além do conteúdo, uma cor diferente que também será pré-definida. Para que cada instrução tenha as duas informações armazenadas, transformaremos as ajudas em uma lista de objetos que têm a propriedade **conteúdo** e **cor**:

```
# js/opcoesDaPagina/btnAjuda.js

const ajudas = [
  "Bem_Vindo_ao_Ceep",
  "Clique_no_btn_Linhas_para_mudar_o_layout"
]
const ajudas = [
```

```

        {conteudo: "Bem Vindo ao Ceep", cor: "#F05450"}
        ,{conteudo: "Clique no botão Linhas para alterar o layout", cor: "#92C4EC"}
    ]

```

2. Com as ajudas novas, podemos já trocar a chamada ao **alert** pelo código que adiciona um cartão no mural. Antes de implementar essa função, é bem importante sabermos como e onde usaremos ela. Para isso, já substituiremos o alert das instruções por uma chamada à função **adicionaCartaoNoMural**, que iremos criar depois.

```
# js/opcoesDaPagina/btnAjuda.js
```

```

alert(ajuda)
adicionaCartaoNoMural(ajuda)

```

3. A função **adicionaCartaoNoMural** ainda não existe, precisamos criá-la. Como adicionar um cartão ao mural é uma funcionalidade que está diretamente ligada ao mural, criaremos essa função em um arquivo **js/mural.js**.

Crie o arquivo **js/mural.js** e inclua ele antes do fechamento da tag **<body>** :

```
# index.html
```

```
<script src="js/mural.js"></script>
```

4. O código da função já foi quase inteiramente criado dentro do arquivo **js/formNovoCartao.js**. É o código que está dentro do **else**, quando verificamos que o usuário digitou algo no **<textarea>** do **.formNovoCartao**. Por enquanto, ele funciona apenas para a funcionalidade de adicionar um novo cartão quando o usuário digita algo através do formulário. Porém, a partir de agora, nossos cartões não virão só desse formulário. Eles também virão das instruções, quando o usuário clicar no botão da ajuda. Assim, para que o código que adiciona um cartão no mural não seja duplicado nesses dois lugares, criaremos e passaremos a usar a função **adicionaCartaoNoMural**. Dentro dela teremos o código que hoje está dentro do **else** lá em **js/formNovoCartao.js**; o que muda, é que a **const conteudoDoCartao** , que antes acessava o texto digitado no **<textarea>** , vai passar a usar a propriedade **conteudo** do **cartaoObj** que recebemos como parâmetro; também adicionaremos um atributo **style** no html do cartão para alterar o **background** , de acordo com a propriedade **cor** do **cartaoObj** .

```
# js/mural.js
```

```

let numeroDoCartao = 0

function adicionaCartaoNoMural(cartaoObj){
    numeroDoCartao++
    const conteudoDoCartao = cartaoObj.conteudo
    const cartao = $(`

<div class="opcoesDoCartao">
            <button class="opcoesDoCartao-remove opcoesDoCartao-opcao" tabindex="0">
                <svg><use xlink:href="#iconeRemover"></use></svg>
            </button>
            <input type="radio" name="corDoCartao${numeroDoCartao}" value="#EBEF40" id="corPadrão-c


```

```

artao${numeroDoCartao}" class="opcoesDoCartao-radioTipo" checked>
    <label for="corPadrão-cartao${numeroDoCartao}" class="opcoesDoCartao-tipo opcoesDoCarta
o-opcao" style="color: #EBEF40;" tabindex="0">
        Padrão
    </label>

    <input type="radio" name="corDoCartao${numeroDoCartao}" value="#F05450" id="corImportan
te-cartao${numeroDoCartao}" class="opcoesDoCartao-radioTipo">
        <label for="corImportante-cartao${numeroDoCartao}" class="opcoesDoCartao-tipo opcoesDoC
artao-opcao" style="color: #F05450;" tabindex="0">
            Importante
        </label>

        <input type="radio" name="corDoCartao${numeroDoCartao}" value="#92C4EC" id="corTarefa-c
artao${numeroDoCartao}" class="opcoesDoCartao-radioTipo">
            <label for="corTarefa-cartao${numeroDoCartao}" class="opcoesDoCartao-tipo opcoesDoCarta
o-opcao" style="color: #92C4EC;" tabindex="0">
                Tarefa
            </label>

            <input type="radio" name="corDoCartao${numeroDoCartao}" value="#76EF40" id="corInspiraç
ão-cartao${numeroDoCartao}" class="opcoesDoCartao-radioTipo">
                <label for="corInspiração-cartao${numeroDoCartao}" class="opcoesDoCartao-tipo opcoesDoC
artao-opcao" style="color: #76EF40;" tabindex="0">
                    Inspiração
                </label>
            </div>
            <p class="cartao-conteudo" contenteditable tabindex="0">${conteudoDoCartao}</p>
        </article>
    `)

// Navegação com focus via teclado nos cartões
cartao.on("focusin", function(){
    cartao.addClass("cartao--focado")
})
cartao.on("focusout", function(){
    cartao.removeClass("cartao--focado")
})

// Funcionalidade muda cor dos cartões
cartao.on("change", ".opcoesDoCartao-radioTipo", function mudaCor(event){
    cartao.css("background-color", event.target.value)
})

cartao.on("keydown", function deixaClicarComEnter(event){
    if(event.target.classList.contains("opcoesDoCartao-opcao") && (event.key === "Enter" || eve
nt.key === " ")){
        event.target.click()
    }
})

//Funcionalidade remove cartão
cartao.on('click', function(event) {
    const elementoSelecionado = event.target
    if(elementoSelecionado.classList.contains('opcoesDoCartao-remove')){
        cartao.addClass("cartao--some")
        cartao.on("transitionend", function(){
            cartao.remove()
        })
    }
})

$(".mural").append(cartao)

```

```
}
```

5. Agora que a função **adicionaCartaoNoMural** está criada, precisamos alterar o arquivo **js/formNovoCartao.js** e chamar a nova função. Dentro do **else**, no lugar onde estava antes os códigos que mudamos para a função **adicionaCartaoNoMural** precisamos colocar:

```
# js/formNovoCartao.js

} else {
    adicionaCartaoNoMural ({conteudo: textarea.value})
}
```

Podemos aproveitar e colocar no mesmo arquivo a funcionalidade de limpar o `<textarea>` depois que o cartão já foi criado. Para isso, precisamos adicionar depois do fechamento do **else**:

```
# js/formNovoCartao.js

// apaga o conteúdo do textarea
textarea.value = ""
```

6. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

EXERCÍCIO: O MODULE PATTERN.

23.1 OBJETIVO

1. A variável `numeroDoCartao` controla qual o cartão que terá a cor alterada quando uma das tags `<label>` for clicada. O `<input type="radio" id="corImportante-cartao${numeroDoCartao}">` será selecionado quando a `<label for="corImportante-cartao${numeroDoCartao}">` for clicada. Fizemos isso, porque os `<input type=radio>` são bem feios e estilizar eles não é uma coisa simples; escondemos esses inputs, e transformamos as labels nos nossos botões de escolher a cor.

O que acontece agora é que a variável `numeroDoCartao` pode ser manipulada por qualquer um dos nossos códigos, inclusive via console. Assim, se tivermos um cartão de número 1 e alterarmos a variável `numeroDoCartao` para ter um valor de 0, o próximo cartão adicionado será um cartão de número 1 novamente. Ao tentar mudar a cor desse segundo cartão, a cor do primeiro é que será alterada.

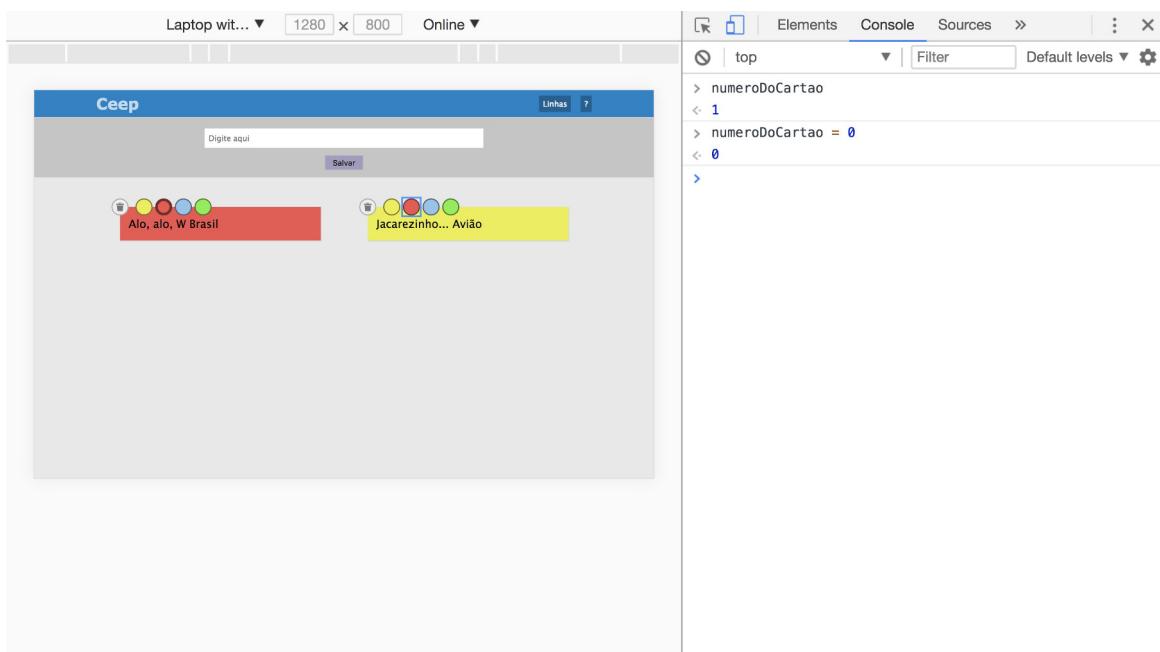


Figura 23.1: Variável `numeroDoCartao` sendo alterada no console

Nosso objetivo é tirar essa variável do escopo global, e no fim, deixar impossível o acesso a essa variável, tanto no console quanto em outros códigos:

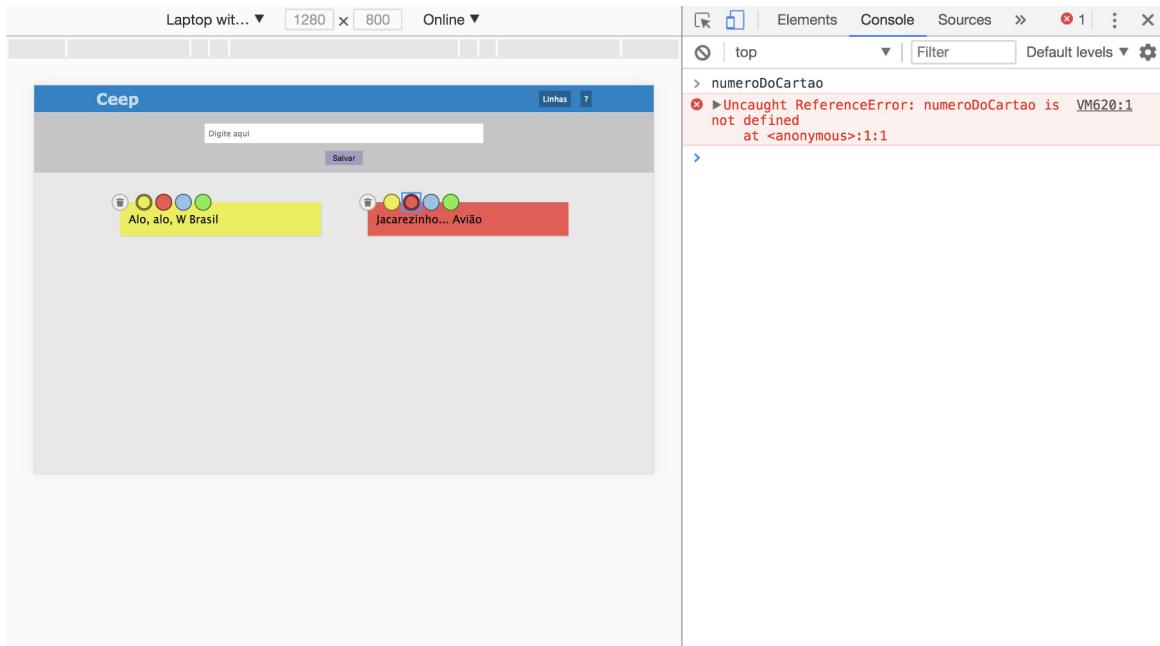


Figura 23.2: Variável numeroDoCartao não existe no escopo global

23.2 PASSO A PASSO COM CÓDIGO

1. Variáveis que não estão dentro de funções são sempre globais para todos os códigos JavaScript rodando no navegador. Para que a variável `numeroDoCartao` não seja global, envolveremos todo o código do arquivo **js/mural.js** numa IIFE. Para impedir que a variável seja criada sem a keyword `let` e se torne global, diremos para o browser que queremos entrar no modo estrito dentro da IIFE.

```
# js/mural.js
```

No início do arquivo, vamos iniciar a IIFE e colocar o "use strict".

```
; (function(){
  "use strict"
```

E na última linha do arquivo, devemos fechar nossa IIFE.

```
)()
```

2. Para que a função `adicionaCartaoNoMural` esteja disponível globalmente, mesmo sendo criada dentro da IIFE, precisaremos exportá-la para o escopo global. No caso do Browser, isso significa passá-la pra dentro do objeto `window`.

```
# js/mural.js
```

```
function adicionaCartaoNoMural(cartaoObj){
  window.adicionaCartaoNoMural = function(cartaoObj){
```

3. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o

seu professor agora.

CAPÍTULO 24

EXERCÍCIO: JAVASCRIPT MODERNO EM NAVEGADORES PRÉ-HISTÓRICOS.

24.1 OBJETIVO

1. Nossa site roda perfeitamente em navegadores modernos, porém se acessarmos ele em um browser mais antigo, como o Firefox 22, nosso código JavaScript não funciona.

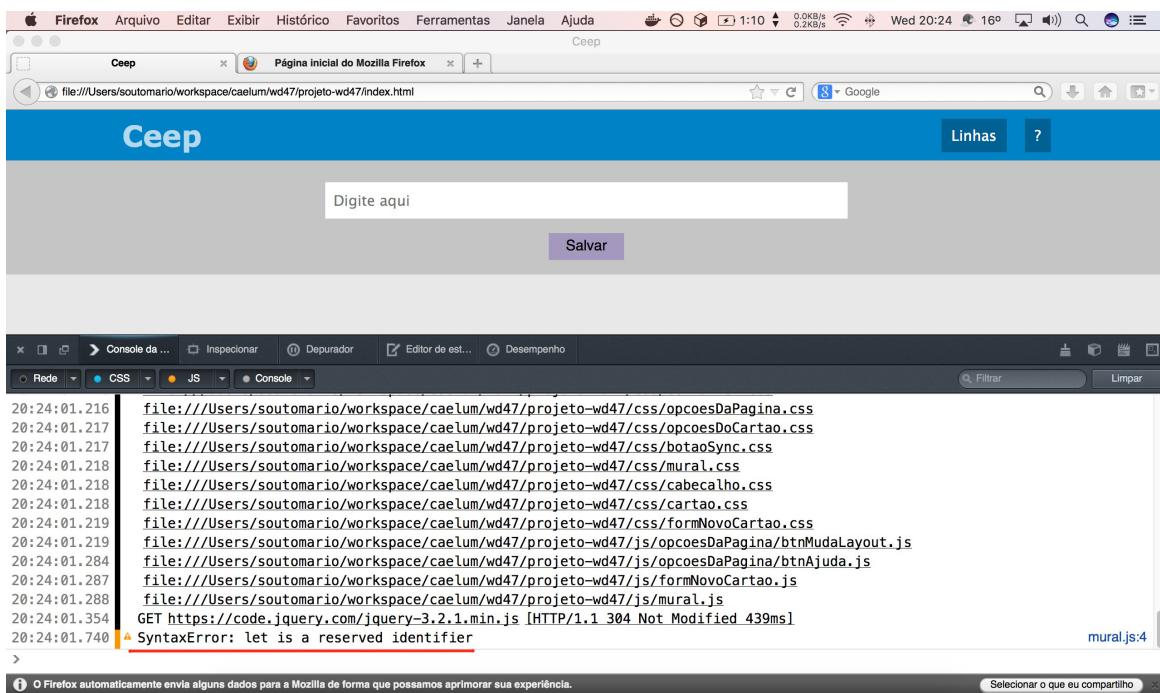


Figura 24.1: Firefox não sabe interpretar as palavras reservadas do JavaScript Moderno

O que acontece aqui é que alguns pontos do nosso código foram feitos usando os recursos mais modernos do JavaScript, como: `let` e a Template String ````. Precisamos fazer com que coisas que **são novas da linguagem** sejam traduzidas para coisas mais antigas da linguagem, que já existiam nesses navegadores mais antigos. Não precisamos escrever nosso projeto com código antigo só para funcionar nesses navegadores; podemos escrever código moderno e automaticamente traduzir esses códigos para abrir nos navegadores.

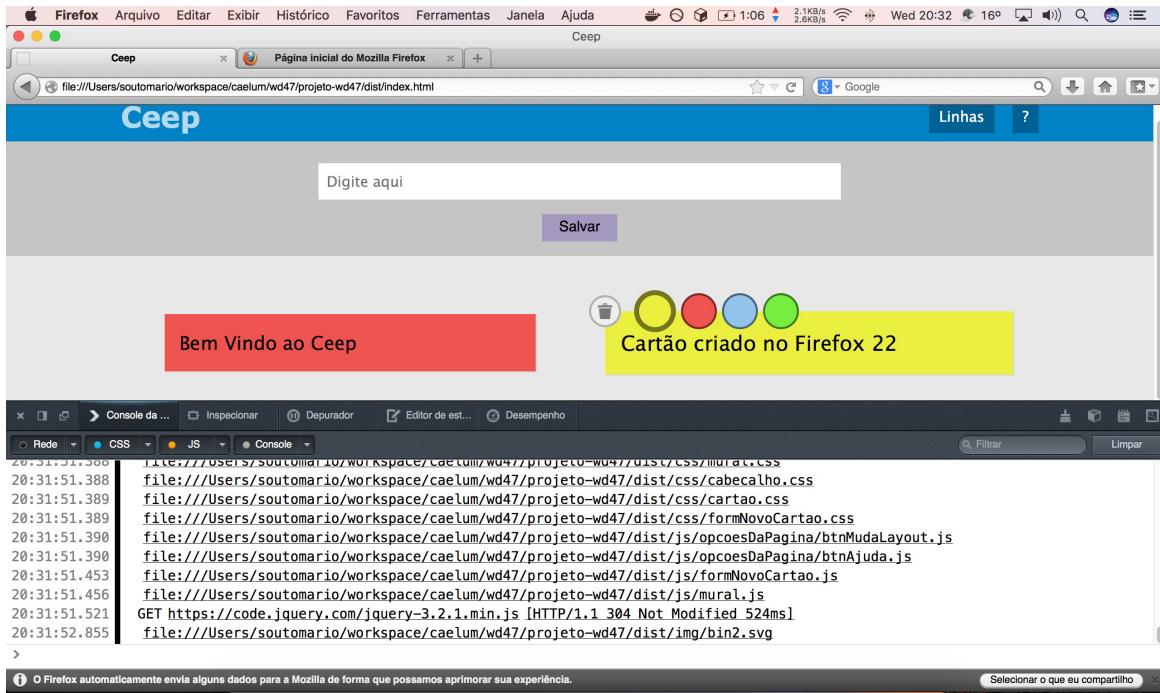


Figura 24.2: Firefox agora não da mais erros e tudo funciona normalmente

24.2 PASSO A PASSO COM CÓDIGO

1. A partir de agora, sempre que escrevermos nosso código, automaticamente geraremos a versão traduzida desse código em outro arquivo. Para separar os códigos que estamos escrevendo dos códigos traduzidos pelo babel, colocaremos todos os nossos códigos numa pasta `src`, onde ficarão os arquivos originais.

Pra começar devemos pegar todos os arquivos da nossa aplicação e colocar numa pasta `src/`.

```
# src/
```

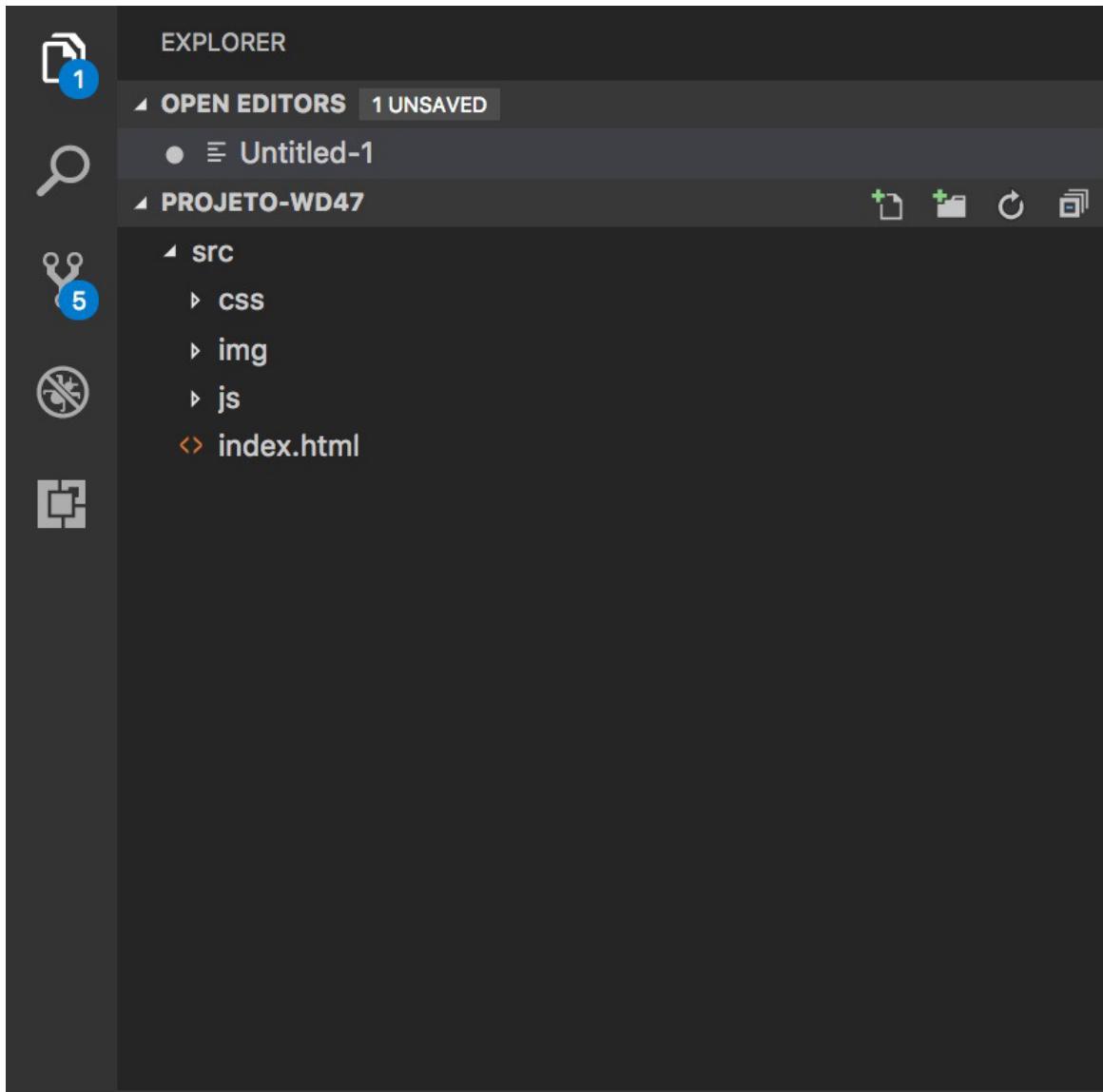


Figura 24.3: Como devem ficar os arquivos do projeto

2. Pelo site babeljs.io é possível traduzir o código JavaScript para a versão que queremos rodar nos browsers antigos. Porém, sempre que alterarmos um arquivo, teremos que copiar o nosso código, colar no site do babel, copiar o que foi gerado e salvar num arquivo. Para facilitar esse processo, iremos baixar o código do babel e configuraremos ele para fazer a tradução automaticamente pra gente.

O babel tem uma **CLI**, ou seja, não tem interface gráfica, roda no **Terminal**. Para isso, precisamos acessar a pasta do projeto via terminal; se o projeto estivesse na pasta CEEP dentro da Área de Trabalho, poderíamos navegar com o seguinte comando:

```
# TERMINAL
```

```
cd ~/Desktop/ceep
```

Após navegar para a pasta do projeto, podemos rodar o seguinte comando para baixar o babel:

```
# TERMINAL
```

```
npm install babel-cli babel-preset-env
```

Nosso próximo passo é rodar o babel:

```
# TERMINAL
```

```
node_modules/.bin/babel src --out-dir dist --copy-files --watch --presets env
```

3. Nesse ponto, nossa pasta **dist/** foi gerada com todos os arquivos traduzidos. Agora sempre que um arquivo for alterado, a versão traduzida será colocada na pasta **dist/**.

PONTOS IMPORTANTES ### A partir de agora, só edite seus códigos na pasta **src/** e abra no navegador os arquivos da pasta **dist/**. Para que nossos próximos códigos sejam transpilados e passados para a pasta **dist/** automaticamente, é importante que o babel sempre esteja rodando enquanto desenvolvemos.

4. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 25

EXERCÍCIO: TRAZENDO AS INSTRUÇÕES COM AJAX.

25.1 OBJETIVO

1. O Ceep está crescendo e agora temos uma equipe de suporte que cuida da elaboração dos textos das instruções. Essa equipe não trabalha com desenvolvimento e cadastra as instruções num sistema separado do que estamos desenvolvendo. Precisamos pegar essas instruções do sistema deles ao invés de deixá-las fixas no arquivo: `src/js/opcoesDaPagina/btnAjuda.js`. As instruções estão disponíveis na seguinte URL: <https://ceep.herokuapp.com/cartoes/instrucoes>. Após trazer a lista de instruções dessa URL via AJAX, é necessário adicionar os cartões na página.

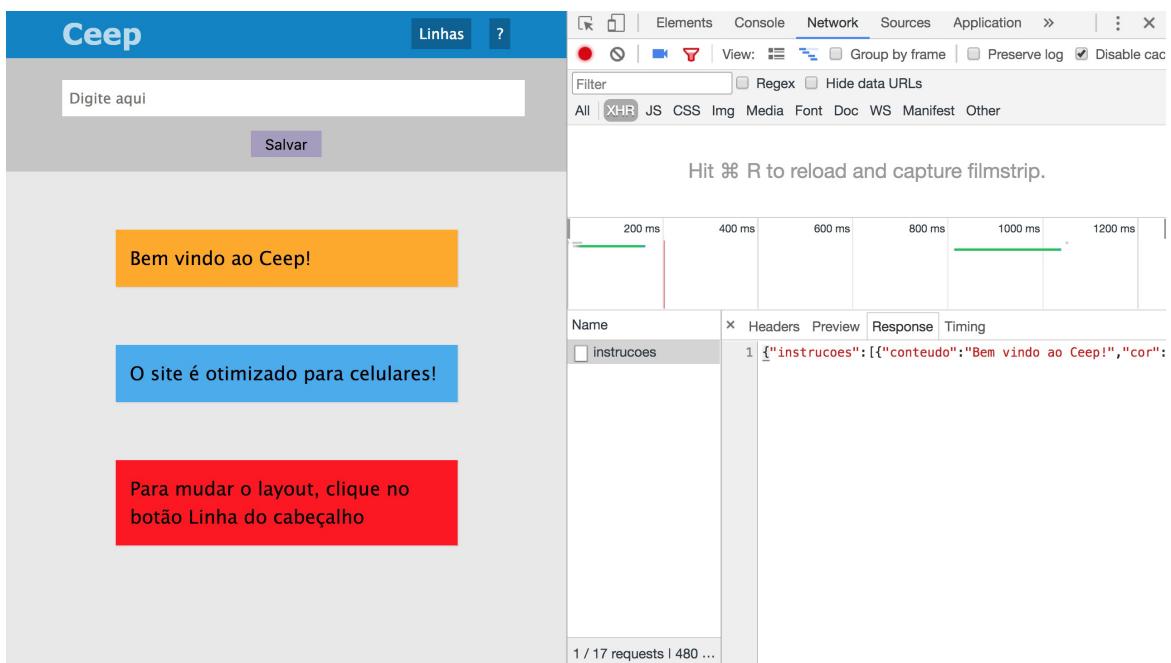


Figura 25.1: Conteúdo dos cartões criado dinamicamente

25.2 PASSO A PASSO COM CÓDIGO

1. O código que cria um cartão pra cada ajuda na lista de ajudas já está pronto no arquivo `src/js/opcoesDaPagina/btnAjuda.js`.

Vamos remover o array de objetos que fixamos no arquivo e fazer uma requisição AJAX para a url

<https://ceep.herokuapp.com/cartoes/instrucoes>, no retorno dessa requisição teremos a lista de ajudas fornecida pela API. Com a lista em mãos, executaremos o mesmo **forEach** que já temos no nosso código.

```
# src/js/opcoesDaPagina(btnAjuda.js
```

O código deve ficar assim:

```
; (function(){
  const btnAjuda = document.querySelector('#btnAjuda')
  btnAjuda.addEventListener("click", function(){
    //pegador de ajudas
    const xhr = new XMLHttpRequest()
    xhr.open('GET', 'https://ceep.herokuapp.com/cartoes/instrucoes')
    xhr.responseType = "json"
    xhr.send()
    xhr.addEventListener("load", function(){
      const objeto = xhr.response
      const ajudas = objeto.instrucoes

      ajudas.forEach(function(ajuda){
        adicionaCartaoNoMural(ajuda)
      })
    })
  })
})()

btnAjuda.classList.remove('no-js')
```

2. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 26

EXERCÍCIO: SALVANDO OS CARTÕES NO SERVIDOR.

26.1 OBJETIVO

1. Sempre que recarregamos a página perdemos nossos cartões. Para contornar esse problema, faremos com que sempre que clicarmos no botão de sincronizar, todos os cartões do mural sejam salvos no servidor do Ceep.

O identificador do usuário e os cartões devem ser enviados para <https://ceep.herokuapp.com/cartoes/salvar> via AJAX. É esperado que o conteúdo da requisição seja do tipo `application/json`, no seguinte formato:

```
{  
  "usuario": "seuemail@email.com.br"  
  , "cartoes": []  
}
```

Após ter os cartões salvos, deve ser possível verificar que as informações foram enviadas na aba **Network** do Dev Tools:

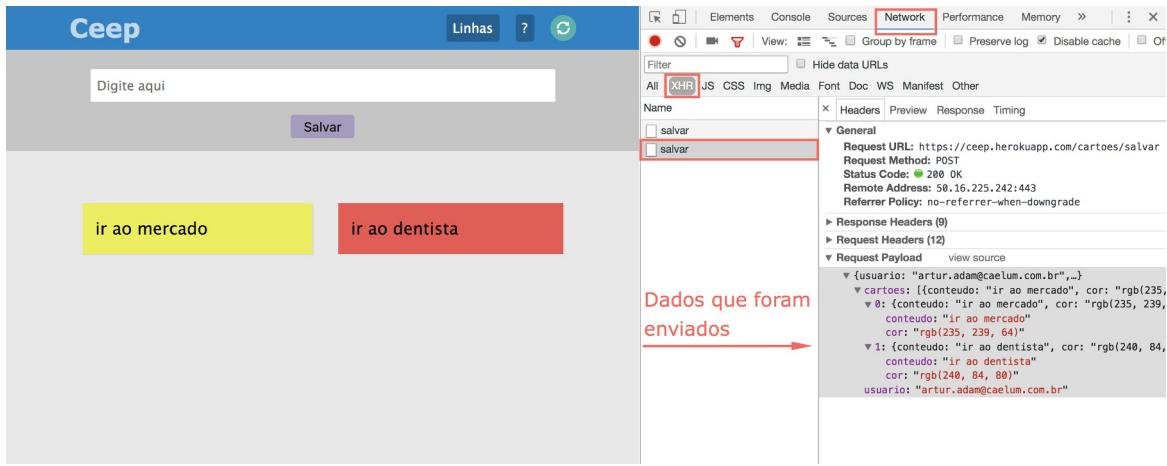


Figura 26.1: Dados da requisição XHR aparecem na aba Network do Dev Tools

No console, também exibiremos a informação de quantos cartões foram salvos no servidor; essa informação estará disponível no formato JSON, na resposta da requisição:

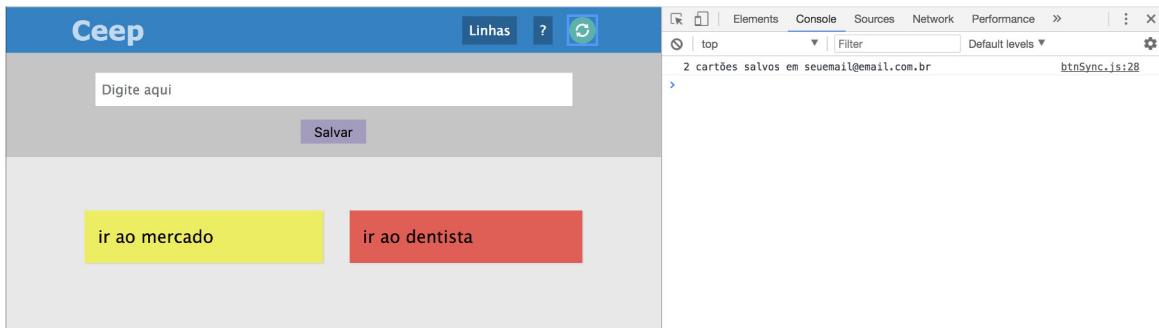


Figura 26.2: Cartões sendo salvos e botão de sync ficando verde para indicar que tudo deu certo

26.2 PASSO A PASSO COM CÓDIGO

1. Toda a lógica de salvar vai ser feita através do click no `<button>` com o id `#btnSync`, por se tratar de uma funcionalidade nova, vamos criar um novo arquivo para isso.

Crie o arquivo **js/opcoesDaPagina/btnSync.js** e inclua ele antes do fechamento da tag `</body>`:

```
# index.html

<script src="js/opcoesDaPagina/btnSync.js"></script>
```

2. Para implementar nossa lógica de salvar os cartões vamos precisar passar por algumas etapas:

- Selecionar o `<button id="#btnSync">` e criar o Event Listener;
- Adicionar as classes que fazem o `<button id="#btnSync">` girar enquanto espera a sincronização terminar
- Iniciar o nosso AJAX com o `new XMLHttpRequest()` desta vez do tipo `POST` apontando para a url `https://ceep.herokuapp.com/cartoes/salvar` e enviando as informações para o servidor através da função `send()`; no nosso caso a informação enviada será um objeto com os cartões do mural e o email do usuário que é dono desse mural.
- Por fim, adicionamos os Event Listeners de `load` e `error`, onde faremos o `<button id="#btnSync">` parar de girar e ficar verde ou vermelho.

```
# js/opcoesDaPagina/btnSync.js

;(function(){
  const btnSync = $("#btnSync")
  btnSync.click(function(){
    btnSync.addClass("botaoSync--esperando")
    btnSync.removeClass("botaoSync--sincronizado")

    const salvadorDeCartoes = new XMLHttpRequest()
    salvadorDeCartoes.open('POST', 'https://ceep.herokuapp.com/cartoes/salvar')
    salvadorDeCartoes.setRequestHeader("Content-Type", "application/json")

    const cartoes = document.querySelectorAll(".cartao")
    const infosDoMural = {
      usuario: "seuemail@email.com.br"
```

```

        ,cartoes: Array.from($(".cartao")).map(function(cartao){
            return {
                conteudo: cartao.querySelector(".cartao-conteudo").textContent
                ,cor: getComputedStyle(cartao).getPropertyValue("background-color")
            }
        })
    }

salvadorDeCartoes.send(JSON.stringify(infosDoMural))

salvadorDeCartoes.addEventListener("load", function(){
    const response = JSON.parse(salvadorDeCartoes.response)
    console.log(`${response.quantidade} cartões salvos em ${response.usuario}`)

    btnSync.removeClass("botaoSync--esperando")
    btnSync.addClass("botaoSync--sincronizado")
})

salvadorDeCartoes.addEventListener("error", function(){
    btnSync.removeClass("botaoSync--esperando")
    btnSync.addClass("botaoSync--deuRuim")
})
}

btnSync.removeClass('no-js')
})()

```

3. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 27

EXERCÍCIO: CARREGANDO OS CARTÕES DO SERVIDOR.

27.1 OBJETIVO

1. Nossos cartões são salvos no servidor, mas ainda assim não é possível visualizá-los no mural quando carregamos a página. Precisamos fazer com que as informações do usuário que se encontram na url <https://ceep.herokuapp.com/cartoes/carregar> sejam carregadas.

O servidor espera uma requisição GET com um parâmetro chamado **usuario**, para identificar quais cartões devem ser carregados, e devolve os cartões no formato **JSONP**.



Figura 27.1: Cartões agora são carregados assim que a página carrega

27.2 PASSO A PASSO COM CÓDIGO

1. Precisamos fazer uma requisição AJAX do tipo GET para pegar os conteúdos. A URL que carrega os cartões não é habilitada para receber requisições desse tipo vindas de qualquer lugar. Para isso, vamos utilizar a estratégia do JSONP que o jQuery nos oferece pronta; chamaremos a função `$.ajax()` passando no objeto de parâmetros a propriedade `dataType` com valor `jsonp`.

```
# src/js/mural.js
```

Esse código deve ser inserido antes do fechamento da IIFE no final do arquivo:

```
$.ajax({  
  url: "https://ceep.herokuapp.com/cartoes/carregar"
```

```
,method: "GET"
,data: {usuario: "seuemail@email.com.br"}
,dataType: "jsonp"
,success: function (objeto){
    const cartoes = objeto.cartoes
    cartoes.forEach(function(cartao){
        adicionaCartaoNoMural(cartao)
    })
}
})
```

2. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

CAPÍTULO 28

EXERCÍCIO: COMPATIBILIDADE NAS APIs DO JAVASCRIPT COM POLYFILL.

28.1 OBJETIVO

1. Para fazer a conversão do `jQuery Object` para um `Array` usamos a função `Array.from`, porém ela não funciona em navegadores antigos. Nesses casos, podemos nós mesmos implementar a função, porém, diversas outras APIs que utilizamos durante o curso podem não funcionar dependendo da versão do navegador sendo utilizada.

Os polyfills são códigos JavaScript que implementam as APIs especificadas pela W3C nos navegadores que não têm essas APIs implementadas, ou nos navegadores que as implementam de maneira errada. Várias dessas APIs já têm polyfills prontos e criados pela comunidade. Podemos adicionar os arquivos JavaScript de cada polyfill que acharmos que precisamos. Porém, isso pode mudar de usuário para usuário. Por isso, usaremos o serviço `polyfill.io` que gera esses polyfills automaticamente dependendo do navegador do usuário. Ele faz a análise do `User Agent` do usuário e cria um arquivo `js` com todos os polyfills necessários.

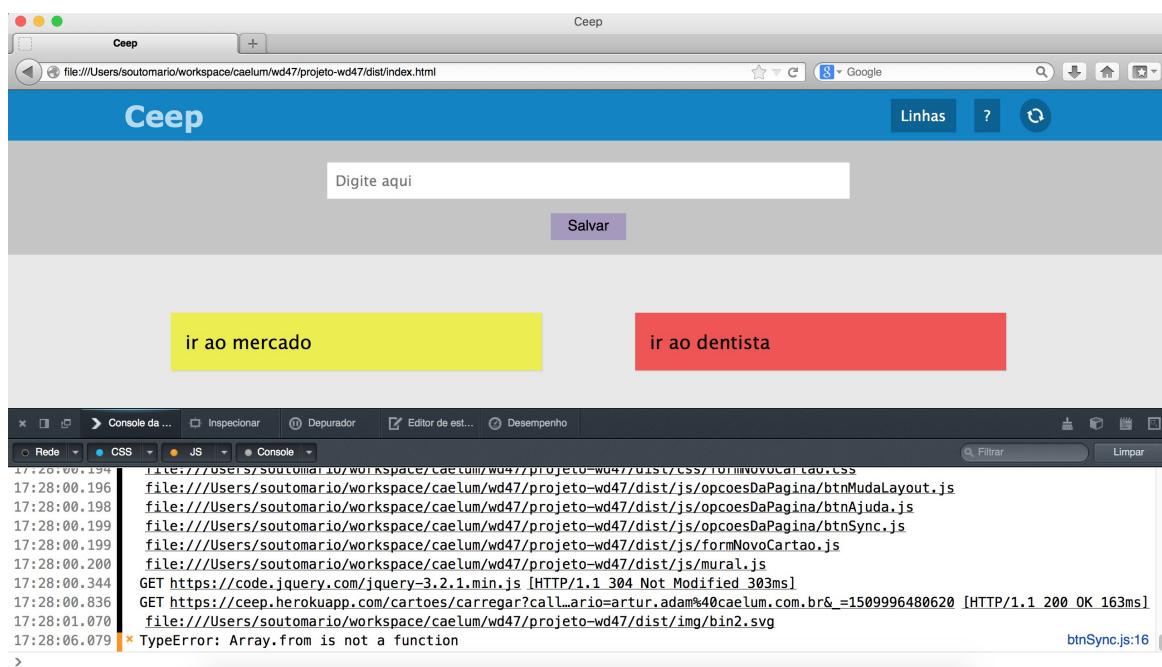


Figura 28.1: Console do Firefox v22 mostra que `Array.from` não existe

28.2 PASSO A PASSO COM CÓDIGO

1. Tudo o que precisamos fazer é incluir uma tag `<script>` na página com o atributo `src` apontando para o serviço do **polyfill.io** .

```
# index.html
```

```
<script src="https://cdn.polyfill.io/v2/polyfill.min.js"></script>
```

Você pode ver mais sobre o serviço dos polyfills no site <https://polyfill.io/v2/docs/>.

2. Não se esqueça de testar o resultado no browser e verificar se o objetivo desse exercício está funcionando. Esse também é o momento de você validar se aprendeu, se ficou qualquer dúvida chame o seu professor agora.

APÊNDICE - DANDO PODERES AO CONTEÚDO

29.1 TRANSFORMANDO TEXTOS EM OUTROS TEXTOS

Nosso usuário já consegue adicionar e remover os cartões da nossa página. Porém, ainda não consegue escrever cartões com quebras de linha.

Qualquer tipo de espaço e quebra de linha que não esteja apenas separando uma palavra ou parágrafo é ignorado pelo html.

Se uma quebra de linha é importante, ela deve ser marcada com um `
`.

O conteúdo que obtemos do `<textare>` através da função `val()` precisa ser modificado. Precisamos checar a presença de caracteres "`\n`" e trocá-los por uma tag `
`.

29.2 STRING.REPLACE()

Qualquer manipulação de texto que envolva trocar um pedaço do texto por outro pode ser feito por meio da função `replace`.

```
var conteudo = "Esse é o conteudo do cartão.\n Acabei de pular uma linha".
conteudo.replace("\n", "<br>")
//resultado é "Esse é o conteudo do cartão.<br> Acabei de pular uma linha".
```

No caso acima, em que temos apenas um "`\n`", o `replace` acontece com sucesso. Num caso com mais quebras de linha, o resultado é diferente:

```
var conteudo = "Conteudo do cartão.\n Pulei uma linha\n Pulei outra linha".
conteudo.replace("\n", "<br>")
//resultado é "Conteudo do cartão.<br> Pulei uma linha\n Pulei outra linha".
```

Não queremos trocar apenas 1 carácter. Precisamos dizer ao javascript que para qualquer caractere "`\n`" efetuaremos a troca. Precisamos mostrar ao javascript um **padrão de caracteres** a serem trocados.

29.3 EXPRESSÃO REGULAR EM JAVASCRIPT

No JavaScript, podemos procurar a por um padrão e substituir os caracteres resultantes por uma outra String. Para representar um padrão de caracteres, utilizamos Expressões Regulares.

No JavaScript, há suporte nativo a expressões regulares.

O assunto é bastante amplo, por isso nosso foco será demonstrar sua integração com JavaScript. Caso você queira saber um pouco mais pode consultar <http://www.regular-expressions.info/>.

Criando expressões regulares

Assim como criamos strings, a criação de expressões regulares é bem parecida:

```
var padrao = /sua expressao regular/;
```

Repare que somos obrigados a começar com `/` e a terminar também com `/`.

Testando a existência de um padrão

Podemos testar se uma String possui o padrão procurado através da função `test`, que retorna `true` ou `false`:

```
var frase = "Quanto é 10 mais 20?";
var padrao = /\d/;

padrao.test(frase); // true
```

No exemplo acima, utilizando a sintaxe especial da expressão regular, testamos se a variável `frase` possui algum dígito, representado pelo padrão `\d`.

Funções de String que aceitam expressão regular

Algumas funções do JavaScript recebem como parâmetro uma expressão regular. Um exemplo é a função `match` do objeto `String`, que retorna um `array` com strings que atendem o critério passado como parâmetro:

```
var frase = "Quanto é 10 mais 20?";
var ocorrencias = frase.match(/\d/); // retorna um array de um elemento ["1"]
```

O resultado da função `match` acima retorna apenas um array de um elemento. Repare que, na string, o padrão `/\d/` aparece mais de uma vez, mas a função parou imediatamente na primeira ocorrência.

Para encontrar todas as ocorrências de um determinado padrão, podemos passar o parâmetro "g" (global) para a expressão regular:

```
var frase = "Quanto é 10 mais 20?";

var ocorrencias = frase.match(/\d/g);
// retorna um array com quatro elementos["1", "0", "2", "0"].
```

Repare que o resultado acima não é o esperado, pois ele considerou cada dígito dos números individualmente. Se um ou mais dígitos forem necessários, utiliza-se o operador `+`. Ele indica para a expressão regular para considerar os dígitos seguintes enquanto o resultado for um número:

```

var frase = "Quanto é 10 mais 20?";
var ocorrencias = frase.match(/\d+/g);
// retorna um array com dois elementos["10", "20"].

```

29.4 EXERCÍCIO: CARTÕES MAIS PODEROSOS COM EXPRESSÕES REGULARES

1. Conseguimos adicionar os cartões, porém, nenhuma quebra de linha é respeitada. Trocaremos todos os \n por
 , no momento da criação do cartão.

```

// esse Event Listener já existe, fizemos ele no exercício anterior
$(".novoCartao").submit(function(event){
    var conteudo = campoConteudo.val().trim()
        .replace(/\n/g, "<br>");
})

```

2. (opcional) Quando algum texto for marcado ****dessa forma**** queremos que ele se torne um texto em negrito: dessa forma . Crie um replace com uma Expressão Regular que seja capaz disso.
3. (desafio) Quando algum texto for marcado ***dessa forma*** queremos que ele se torne um texto em itálico: dessa forma . Crie um replace com uma Expressão Regular que seja capaz disso.

29.5 MEDIDAS RELATIVAS: EM

A grande diferença de `em` e `px` é que `em` é uma **medida relativa**. O valor é calculado levando sempre em consideração o `font-size` do pai. Isso quer dizer que um elemento com `font-size: 2em;` vai ter o dobro do tamanho da fonte do pai, seja ele qual for.

E assim por diante. O pai do pai do pai do pai... É uma grande multiplicação de valores até chegar na raiz do documento, o `<html>` . Se não tiver uma fonte definida, a maioria dos navegadores usa 16px como padrão para o `<html>` .

Isso quer dizer que esse código:

```

<html>
  <body>
    <main style="font-size: 1.5em">
      <h1 style="font-size: 2em">Titulo</h1>
      <p style="font-size: 0.75em">Texto.</p>
    </main>
  </body>
</html>

```

...acaba sendo equivalente a:

```

<html style="font-size: 16px">
  <body style="font-size: 16px">
    <main style="font-size: 24px">
      <h1 style="font-size: 48px">Titulo</h1>
    </main>
  </body>
</html>

```

```
<p style="font-size: 18px">Texto.</p>
</main>
</body>
</html>
```

Onde o em faz sentido?

A grande vantagem do `em` é o seu aspecto de multiplicar os valores de acordo com os pais. Isso quer dizer que você pode mudar o `font-size` de um elemento e isso afetar todos os seus filhos. É muito útil para criar seções na página onde os elementos devem aumentar proporcionalmente entre si – um componente, um widget.

Ou seja, usamos `em` para facilitar a escrita do nosso CSS. E por isso é bom dominar o uso de `em` e saber aproveitá-lo no seu código.

O `em`, claro, não é para todos os cenários. Ele vincula as medidas do filho com o `font-size` do pai. Você mexe no pai e o filho é afetado. Isso às vezes é indesejado. Só usamos `em` quando existe uma relação estrutural entre o filho e o pai e queremos que um afete o outro. Onde não fizer sentido, continue usando `px`.

O caso famoso dos designs responsivos

O `em` tem um uso forte nos designs responsivos. É que é muito frequente você querer uma fonte menor no mobile, que tem menos espaço visível, mas uma fonte maior no Desktop, mais espaçoso. E você quer que a página toda aumente proporcionalmente. Por isso o `em`.

Fazemos algo assim:

```
@media (min-width: 500px) {
  html {
    font-size: 1.25em;
  }
}
@media (min-width: 800px) {
  html {
    font-size: 1.5em;
  }
}
@media (min-width: 1000px) {
  html {
    font-size: 1.75em;
  }
}
@media (min-width: 1200px) {
  html {
    font-size: 2em;
  }
}
```

Conforme o tamanho da tela aumenta, aumentamos o valor do `em` base na tag

. Isso causa um efeito cascata em todos os elementos filhos se estiverem escritos com `em`.

É muito bom usar `em` nos sites responsivos para ajustar, via CSS, o tamanho de todos os elementos proporcionalmente.

Mas `em` não é uma medida flexível nem mais acessível. Pelo menos não no sentido de outras como a porcentagem, que se adapta automaticamente. O `em` é fixo, só o valor que é calculado a partir de uma conta mais complicada.

O REM

O `rem` é uma medida nova parecida. A diferença é que a conta não leva em consideração todos os pais, mas apenas a raiz, o `<html>`. Isso quer dizer que mexer no `font-size` de algum elemento na página não vai refletir e atrapalhar outros. Só se mexermos no `<html>` mesmo. É útil para o caso do design responsivo que vimos antes.

Mas lembre que o `em` vai multiplicar pelo `font-size` do pai, o que é uma coisa boa em muitas situações. Várias partes da página podem ser encaradas como pequenos componentes autocontidos. E mexer no pai do componente tem que afetar todos os filhos. O `em` é excelente pra isso, ao contrário do `rem`.

29.6 ALINHAMENTO COM FLEXBOX

Quando dispomos os elementos flex no container, podemos controlar como eles se alinham no eixo flex. Por exemplo, quando temos o padrão, de dispor os elementos na linha (`row`), podemos controlar se eles se alinham pelo topo da linha, pelo centro, por baixo etc.

Isso é feito com a propriedade `align-items`. Por exemplo, para colocar os elementos alinhados pelo topo da linha, usariamos:

```
.mural {  
    align-items: flex-start;  
}
```

Além de `flex-start`, temos `flex-end`, `center` e outros. Essa imagem ajuda a entender:

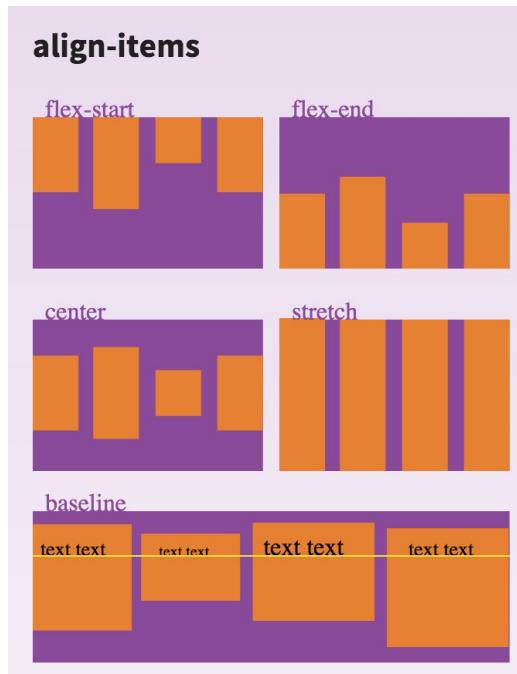


Figura 29.1: Opções do align-items, by CSStricks

O padrão é o valor `stretch`. É importante saber também que o `align-items` muda a disposição dos elementos no eixo principal do flex. Isso quer dizer que se estamos trabalhando com `row` (padrão), ele diz respeito ao alinhamento vertical dos filhos; se fosse `column`, seria o alinhamento horizontal.

29.7 O FOREACH DO ES5

Durante muito tempo, o JavaScript só possuía o `for` e o `while` normais para iterar em listas de elementos. O jQuery até trazia uma função `forEach` para facilitar a iteração, algo comum em outras linguagens e ambientes funcionais.

A versão 5 do EcmaScript, base do JavaScript, adicionou o `forEach` nativamente na linguagem. Então hoje é possível escrever códigos assim:

```
var lista = [1, 5, 4, 3, 7];

lista.forEach(function(elemento){
    console.log(elemento);
});
```

O ponto ruim é que não funciona em todos os navegadores. Mas há um bom suporte, desde o IE9 por exemplo. Caso você queira suportar navegadores mais antigos, é fácil implementar essa função onde ela não existir. Consulte:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach#Compatibilidade

29.8 EXERCÍCIO: MELHORANDO A VIZUALIZAÇÃO DOS CARTÕES

- Teremos três categorias de tipos de cartões: com texto pequeno, com texto médio e com texto grande. Esses tipos de cartões devem alterar o `font-size`, `width` e `flex-basis` dos cartões em telas maiores. Adicione em `cartao.css`:

```
@media (min-width: 560px){  
  
    .cartao--textoPequeno {  
        font-size: 1em;  
        width: 11em;  
        flex-basis: 11em;  
    }  
  
    .cartao--textoMedio {  
        font-size: 1.5em;  
        width: 9em;  
        flex-basis: 9em;  
    }  
  
    .cartao--textoGrande {  
        font-size: 2em;  
        width: 6em;  
        flex-basis: 6em;  
    }  
  
}
```

Para testar, adicione algumas dessas classes nos cartões de teste.

- Para melhorar a legibilidade dos cartões, aumentaremos o tamanho da fonte dependendo da largura da tela. Esses estilos serão aplicados no `mural`, por isso, criaremos um arquivo `mural.css` só para ele, na pasta `css`.

```
.mural {  
    font-size: 1.2rem;  
}  
  
@media (min-width: 610px){  
    .mural {  
        align-items: flex-start;  
    }  
}  
  
@media (min-width: 900px){  
    .mural {  
        align-items: center;  
        font-size: 1.3rem;  
    }  
}
```

- Criaremos agora a função que nos dirá qual o tipo do nosso cartão. Primeiramente, precisamos contar o número de quebras de linha, o total de letras e o tamanho da maior palavra. Depois, baseados nesses dados, decidir qual categoria dar ao cartão.

```

function decideTipoCartao(conteudo){
    var quebras = conteudo.split("<br>").length;

    var totalDeLetras = conteudo.replace(/<br>/g, " ").length;

    var ultimoMaior = "";
    conteudo.replace(/<br>/g, " ")
        .split(" ")
        .forEach(function(palavra){
            if (palavra.length > ultimoMaior.length) {
                ultimoMaior = palavra;
            }
        });
    var tamMaior = ultimoMaior.length;

    //no mínimo, todo cartão tem o texto pequeno
    var tipoCartao = "cartao--textoPequeno";

    if (tamMaior < 9 && quebras < 5 && totalDeLetras < 55) {
        tipoCartao = "cartao--textoGrande";
    } else if (tamMaior < 12 && quebras < 6 && totalDeLetras < 75) {
        tipoCartao = "cartao--textoMedio";
    }

    return tipoCartao;
}

```

4. Agora, para mudar os tipos dos cartões, chamaremos nossa função `decideTipoCartao` passando o conteúdo do cartão. Adicionaremos seu retorno como classe de todo novo cartão. O código a seguir será colocado no Event Listener de `submit` do formulário de novos cartões.

```

$(".novoCartao").submit(function(event){
    //... aqui, código de aulas passadas

    if (conteudo){

        contador++;

        var botaoRemove = $("<button>").addClass("opcoesDoCartao-remove")
            .attr("data-ref", contador)
            .text("Remover")
            .click(removeCartao);

        var opcoes = $("<div>").addClass("opcoesDoCartao")
            .append(botaoRemove);

        // **código novo** chamada para nova função
        var tipoCartao = decideTipoCartao(conteudo);

        // **código novo** adicionando classe no novo cartão
        $("<div>").attr("id", "cartao_" + contador)
            .addClass("cartao")
            .addClass(tipoCartao)
            .append(opcoes)
            .append(conteudoTag)
            .prependTo(".mural");

    }

    //... aqui, código de aulas passadas
});

```

29.9 MAIS FUNÇÕES DO JQUERY

Nosso próximo exercício implementará a busca nos textos do cartão. Teremos um campo que, conforme o usuário vai digitando, vai filtrando apenas os cartões que batem com aquele critério.

Se quisessemos esconder todos os cartões, usando jQuery, seria algo assim:

```
$(".cartao").hide();
```

Ao usar `$(".cartao")` obtemos uma lista de todos os elementos com essa classe. Aí aplicamos o `hide()` que esconde todos os elementos.

O problema é que não queremos esconder todos. Queremos pegar os cartões mas *filtrá-los* para esconder apenas os que baterem com nossa regras.

A função filter

Dada uma lista de elementos jQuery, podemos chamar a função filter para refiná-la. Essa função recebe cada elemento da lista e deve retornar true ou false, indicando se o elemento deve ou não ser usado.

Por exemplo, para filtrar e esconder todos os cartões cujo texto tem exatos 5 caracteres:

```
$(".cartao").filter(function(){
    return $(this).text().length == 5;
}).hide();
```

Vamos usar a função filter no exercício com um exemplo mais complexo para implementar a busca textual dos cartões.

29.10 EXPRESSÕES REGULARES DINÂMICAS

No próximo exercício, criaremos a funcionalidade de busca por cartões. Procuraremos o termo digitado pelo usuário no conteúdo de cada cartão. Queremos que o termo buscado seja procurado sem diferenciar maiúsculas de minúsculas. Assim, não procuramos por um texto, mas por um padrão.

Como visto antes, padrões em javascript são representados por expressões regulares.

```
//conteúdo de um cartão
var conteudo = "Conteúdo do meu cartão"

//termo que o usuário digitou
var termoBuscado = "conteúdo"

var temPalavraConteudo = conteudo.match(/termoBuscado/i)
```

O código anterior não funciona, pois está buscando pela ocorrência de "termoBuscado". Precisamos criar um padrão baseado num valor variável, que vem do usuário. Para isso, criaremos uma expressão regular dinamicamente:

```

var texto = "Conteudo do meu cartão";
var termoBuscado = "conteudo";

var regex = new RegExp(termoBuscado, "i");

var temPalavraConteudo = texto.match(regex);

```

A função `RegExp`, quando invocada com um `new`, constrói uma expressão regular baseada nos seus dois parâmetros. O primeiro é o padrão, em formato de texto. O segundo, as *flags* da regex, no nosso caso, um "i" indicando que queremos uma regex *case-insensitive*.

Se mandarmos imprimir o valor da variável `regex` acima veremos `/conteudo/i`

29.11 EXERCÍCIO: BUSCANDO CARTÕES COM JQUERY

- O Usuário preencherá um campo de texto para fazer a busca. Esse campo ficará dentro do cabeçalho.

```

<header class="cabecalho container">
    <!-- logo aqui -->
    <div class="opcoesDaPagina">

        <input type="search" id="busca" placeholder="busca"
               class="opcoesDaPagina-opcao">

        <!-- botão aqui-->
    </div>
</header>

```

- No principal.js vamos colocar o evento de input.

```

$("#busca").on("input", function(){
    //guarda o valor digitado, removendo espaços extras.
    var busca = $(this).val().trim();

    if(busca.length){
        $(".cartao").hide().filter(function(){
            return $(this).find(".cartao-conteudo")
                .text()
                .match(new RegExp(busca, "i"));
        }).show();
    }else{
        $(".cartao").show();
    }
});

```

Volte ao navegador, digite algo no campo para ver o resultado.

- (desafio) Faça um *highlight* no texto do cartão que bate com a sua busca. (Dica: para poder estilizar só um pedaço de texto, esse pedaço tem que estar dentro de um elemento próprio).

APÊNDICE - AUTOMATIZAÇÃO DE TAREFAS

Mesmo com um projeto pronto para produção ainda é necessário realizar uma série de tarefas. Colocar prefixos no css para ajustar compatibilidade, concatenar arquivos css e js para diminuir o número de requisições do navegador, retirar os espaçamentos desses arquivos para que eles sejam carregados mais rápidos...

Muitas vezes o desenvolvedor realiza essas tarefas manualmente, o que pode demandar muito tempo, além do projeto estar vulnerável a possíveis erros que o desenvolvedor possa cometer.

Para solucionar problemas como esses, foram criadas no mercado ferramentas de construção (build) de projetos como Ant, Gradle e Maven, mas há algumas que nasceram voltada especialmente para programadores front-end: o **Grunt** e o **Gulp** por exemplo.

São ferramentas diferentes. Foram feitos em JavaScript e com grande foco em automatizar tarefas de front-end. Se você, por exemplo, for seguir as boas práticas de performance para sites, já deve se preocupar em minificar CSS e JavaScript ou ainda juntar arquivos para diminuir o número de requests e até fazer CSS sprites. Ou talvez você esteja usando algum pré-processador de CSS como o LESS, SASS ou Stylus.

Aqui no curso vamos ver **Gulp** que é mais simples de usar e tão bom quanto o famoso **Grunt**.

30.1 UM POUCO SOBRE NODE.JS

O Gulp é escrito em JavaScript mas não executa no browser. Ele executa no terminal usando o **Node.js**.

O Node.js é uma ferramenta para execução de código JavaScript fora do navegador. Ele roda em servidores e no nosso terminal. Sua grande vantagem é permitir o uso da linguagem JavaScript fora do browser, facilitando nossa vida de desenvolvedores JavaScript.

O Node.js é baseado na engine V8 do Google Chrome. É o mesmo executor de JS usado dentro do browser do Google, mas disponível em outros ambientes. Para executá-lo, basta ir no terminal e rodar o comando **node**.

Dentro dele, podemos executar qualquer código JavaScript que não dependa de um browser. Por exemplo:

```
> 17 * 43
731
> var curso = "Caelum";
undefined
> console.log(curso);
Caelum
undefined
```

Mas repare que todo código que envolva o browser e o DOM não funciona. Não temos acesso a objetos como `document`, `window` ou `navigator`:

```
> alert("oi")
ReferenceError: alert is not defined
```

Executando um módulo

Imagine que queremos fazer a conversão de textos ****dessa forma**** para um texto **dessa forma** em vários lugares da nossa aplicação. Podemos transformar essa lógica num módulo, vamos chamá-lo de `formatador.js`:

```
var formatadorDeTexto = (function(){

    function emNegrito(texto){
        return texto.replace(/\*\*(.*?\*)\*/g, "<b>$1</b>");
    }

    return{
        emNegrito: emNegrito
    }
})();
```

Como não temos o `window`, precisamos de outra forma de usar um JS externo. Para isso, o node criou a função `require`, e usá-la é bem fácil:

```
require("formatador.js");
```

O problema é que o node mantém os escopos dos arquivos separados, pra evitar poluição de variáveis globais. Isso quer dizer que, mesmo importando o `formatador.js`, a variável `formatadorDeTexto` não estará disponível.

Resolvemos isso com duas mudanças: primeiro, nosso código que chama o `require` deve receber o módulo e salvar em uma variável:

```
var formatadorDeTexto = require("formatador.js");
```

E, depois, o próprio arquivo `formatador.js` precisa estar preparado pra devolver o módulo. No `node.js`, a maneira de fazer isso é atribuindo o módulo a variável `module.exports`:

```
// no final do arquivo formatador.js
module.exports = formatadorDeTexto;
```

Agora podemos voltar ao Node.js, importar nosso módulo e usá-lo normalmente:

```
> var formatadorDeTexto = require("formatador.js");
undefined
> formatadorDeTexto.emNegrito("só **eu** devo ficar em negrito");
"só <b>eu</b> devo ficar em negrito"
```

MESMO CÓDIGO RODANDO NO BROWSER E NO NODE

Para usar o módulo de formatador no Node, precisamos colocá-lo na variável `module.exports`. Mas essa variável não existe no browser e vai fazer nosso código dar erro. Inclusive, no browser, ela nem precisa existir já que o módulo já é global.

Uma forma de deixar nosso módulo compatível tanto com browser quanto com Node é colocar a exportação do módulo em um if:

```
if (typeof module !== "undefined") {
  module.exports = formatadorDeTexto;
}
```

30.2 INSTALANDO GULP

Para usar o Gulp é necessário ter o **Node.js** e o **npm** (*node package manager*) instalados.

Instalando Node.js

Você pode baixar os instaladores do Linux, Mac e Windows na própria página do `Node.js` em <http://nodejs.org/download/>

ATUALIZANDO NODE JS VIA NPM

Podemos atualizá-lo facilmente via `npm`, se necessário (cheque sua versão atual do Node com o comando `node -v`). Para isso, basta executar os seguintes comandos:

```
npm config set registry http://registry.npmjs.org
npm cache clean -f
npm install -g n
n stable
```

Instalando o Gulp

Com o `Node.js` instalado, utilize o gerenciador de pacotes `npm` na pasta do seu projeto para instalar o Gulp:

```
npm install gulp
```

Dentro da pasta do projeto, será criada a pasta **node_modules** com o **gulp** e todas as suas dependências.

GULP GLOBAL

Para que tenhamos acesso ao gulp na linha de comando A instalação globalmente é recomendada:

```
npm install -g gulp
```

Caso não seja possível instalar o **gulp** globalmente, você pode executar o **gulp** com o comando:

```
./node_modules/.bin/gulp
```

30.3 GULPFILE E TASKS

O Gulp é uma ferramenta de build e automatização onde você escreve suas tarefas (tasks) em JavaScript. Cada funcionalidade é um plugin diferente que você pode instalar e importar no seu código.

O build é escrito todo em um arquivo de script chamado **gulpfile.js** que deve ficar na raiz do seu projeto. Nele, importamos os módulos necessários e depois vamos escrevendo as tasks. O módulo essencial, claro, é o próprio **gulp**:

```
var gulp = require("gulp");

gulp.task("oi", function(){
  console.log("Oi Mundo!");
});
```

Repare como usamos **require** no Node.JS para importar um módulo. E aí temos a API do Gulp com a definição das tarefas, as **tasks**. Cada task recebe um nome e uma função com o código que ela deve executar. No exemplo, não muito útil, apenas mostramos uma mensagem na saída.

Para executar é muito simples. Dado que seu código está no **gulpfile.js** na raiz do projeto, basta executar:

```
gulp oi
```

Ou seja, o comando **gulp** e o nome da task. Nas próximas sessões, veremos como fazer tasks mais úteis.

30.4 EXERCÍCIO: INSTALANDO GULP E A PRIMEIRA TASK

1. Abra o terminal e vá até a pasta do seu projeto, usando o comando **cd**:

```
$ cd Desktop/app
```

2. Agora vamos usar o npm para instalar o gulp, digite:

```
$ npm install gulp
```

3. Crie um arquivo chamado **gulpfile.js** na pasta root do seu projeto. Dentro dele use o `require` para usar o gulp e criar a primeira task

```
var gulp = require("gulp");

gulp.task("oi", function(){
    console.log("Oi Mundo!");
});
```

4. Volte ao terminal e rode a task do gulp:

```
$ gulp oi
```

30.5 PREFIXOS AUTOMÁTICOS

Uma dificuldade grande ao usar vários recursos novos do CSS é lidar com prefixos para diferentes navegadores. Saber qual navegador usa qual prefixo em qual propriedade. Pior, muitas vezes a propriedade muda de nome dependendo da versão. (como o flexbox que é oficialmente `display:flex` mas já foi `display:flexbox` por exemplo)

Logo você percebe que lidar manualmente com todos esses prefixos é uma tarefa bastante complicada. Precisamos automatizar.

O projeto **Autoprefixer** é uma ferramenta que coloca prefixos no seu código CSS automaticamente e ainda sabe declarar sintaxes antigas quando necessário (como o caso do flex).

Ele se integra ao Caniuse.com e gera prefixos para as versões de cada browser que você quiser. Por padrão, ele se preocupa com as 2 últimas versões de cada navegador, e todos os outros que tenham pelo menos 1% de market share.

Usando o autoprefixer na mão

O autoprefixer é um módulo node que você pode instalar e chamar na linha de comando. Passa o seu CSS e ele adiciona os prefixos no seu arquivo.

```
npm install autoprefixer
autoprefixer arquivo.css
```

Usando autoprefixer com gulp

É um tanto trabalhoso rodar o autoprefixer para cada arquivo do projeto. Pior ainda, rodar toda hora que mexermos em algum CSS. Para automatizar essa execução, usamos o Gulp.

Basta instalar o plugin do gulp para o autoprefixer:

```
npm install gulp-autoprefixer
```

No gulpfile, importamos o módulo:

```
var autoprefixer = require("gulp-autoprefixer");
```

Agora precisamos criar a task que execute o autoprefixer para nós. Usando a sintaxe do gulp, vamos encadeando as chamadas e transformações nos arquivos. Importante é saber a origem dos arquivos (`src`) e o destino dos arquivos (`dist`):

```
gulp.task("prefix", function(){
  return gulp.src("css/*.css")
    .pipe(autoprefixer())
    .pipe(gulp.dest("css/"));
})
```

PIPE()

A função `pipe` que vimos acima é referente a um conceito de **encadeamento**, muito comum no terminal. É como colocar comandos numa linha de produção: o segundo comando vai usar o resultado do primeiro, o terceiro vai usar o resultado do segundo, e assim por diante. Para isso, usamos o caracter | (pipe).

```
comando1 | comando2 | comando3
```

Repare que no `src` passamos todos os arquivos `.css` da pasta `css/`. Podemos passar outros caminhos e até gerá-los dinamicamente. É tudo JavaScript.

Então encadeamos uma chamada ao `autoprefixer()`, o plugin que importamos. Ele vai agir na lista de arquivos carregada anteriormente. No fim, escrevemos os arquivos transformados de volta na pasta `css/`.

Para rodar essa task:

```
gulp prefix
```

30.6 COPIANDO ARQUIVOS

Não é recomendado fazer nosso build mexer nos arquivos originais que escrevemos. No caso dos prefixos, ele vai poluir todo nosso `.css` tirando a grande vantagem do autoprefixer de deixar os prefixos pra depois, e escrever css limpo.

A boa prática é ter uma pasta com nossos arquivos originais e copiar tudo para uma nova pasta antes de modificar os arquivos.

Por exemplo, colocar o código original numa pasta `src/` e deixar os arquivos de build numa pasta `dist/`. Podemos fazer isso com Gulp nativamente, apenas indicando uma pasta diferente para o `src()` e o `dest()`:

```
gulp.task("copy", function(){
    return gulp.src("src/**/*").pipe(gulp.dest("dist"));
});
```

Criamos uma nova task `copy` que pega todos os arquivos e subpastas de `src/` e copia para a pasta `dist/`.

Agora podemos rodar com `gulp copy`. E precisamos mudar nossa task de prefixos para agir apenas na pasta `dist/`.

30.7 EXERCÍCIO: COPY E O AUTOPREFIXADOR

1. Separe o projeto do ambiente de desenvolvimento e de produção em duas pastas separadas dentro de `app`, criando a pasta `src` e copiando todo o projeto original para dentro dela. Esse é o ambiente de desenvolvimento. A pasta `dist` será nosso ambiente de produção.
2. Vamos criar uma nova task para copiar os arquivos do nosso projeto. Crie-a no `gulpfile.js` usando as funções `src` e `dest` do gulp:

```
gulp.task("copy", function(){
    return gulp.src("src/**/*").pipe(gulp.dest("dist"));
});
```

3. No terminal rode o comando `gulp copy`. Procure pela nova pasta `dist` e veja se deu tudo certo.
4. Agora vamos fazer funcionar o prefixador. Comece instalando o plugin do autoprefixer no projeto usando o npm no terminal:

```
$ npm install --save-dev gulp-autoprefixer
```

Vá para o `gulpfile.js`. Precisamos usar o `required` para pegar o autoprefixer.

```
var prefixer = require("gulp-autoprefixer");
```

5. Agora vamos criar a tasks para colocar os prefixos. Nossa aplicação deve se compatível com as duas últimas versões de todos os navegadores e O Internet Explorer a partir do 10.

Fique atento aos caminhos de origem e destino e com as funções pipe

```
gulp.task("prefix", function(){
    return gulp.src("dist/css/*.css")
        .pipe(prefixer({
            browsers: ["last 2 versions", "IE 10"]
        }))
        .pipe(gulp.dest("dist/css"));
});
```

6. No terminal, rode o comando `gulp prefix`. Depois de terminado, abra a pasta `css` dentro da pasta

dist e veja o resultado.

30.8 DEPENDÊNCIAS EM TASKS E A TASK DEFAULT

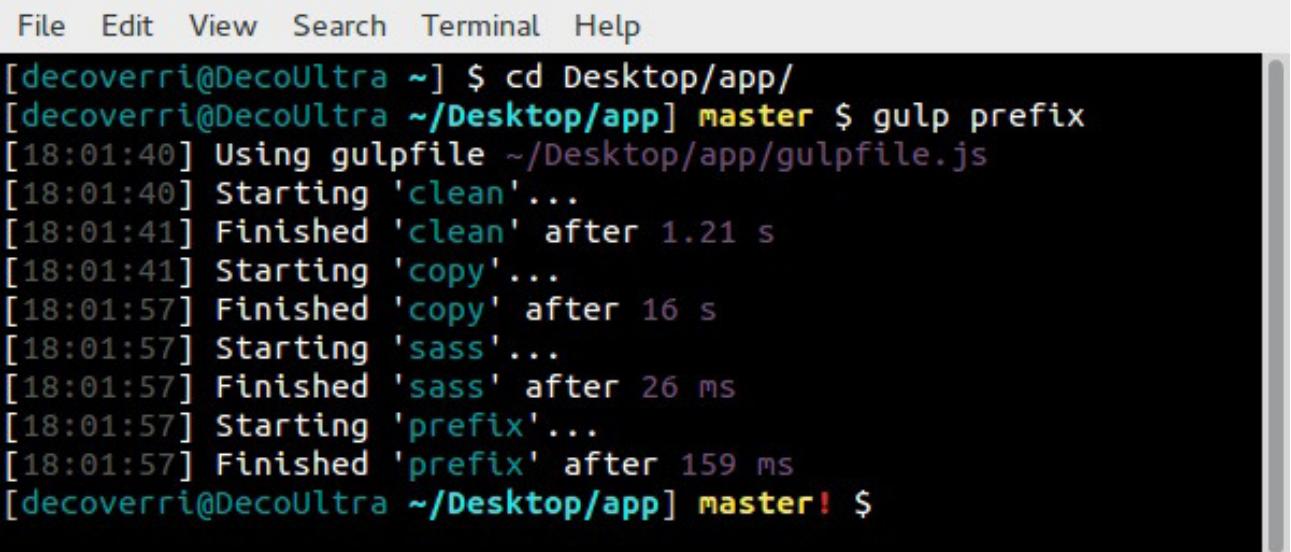
Fizemos duas tasks até o momento: a que copia tudo para uma nova pasta e a que roda o autoprefixer. Rodamos ambas independentemente na linha de comando.

Mas existe uma relação de dependência aqui, uma ordem. A task copy precisa rodar antes que a task dos prefixos.

Com gulp, podemos declarar dependências das tasks no momento de sua declaração:

```
gulp.task("prefix", ["copy"], function(){
  //...
});
```

Agora, só precisamos digitar o comando `gulp prefix` e ele vai chamar as outras tasks.



```
File Edit View Search Terminal Help
[decoverri@DecoUltra ~] $ cd Desktop/app/
[decoverri@DecoUltra ~/Desktop/app] master $ gulp prefix
[18:01:40] Using gulpfile ~/Desktop/app/gulpfile.js
[18:01:40] Starting 'clean'...
[18:01:41] Finished 'clean' after 1.21 s
[18:01:41] Starting 'copy'...
[18:01:57] Finished 'copy' after 16 s
[18:01:57] Starting 'sass'...
[18:01:57] Finished 'sass' after 26 ms
[18:01:57] Starting 'prefix'...
[18:01:57] Finished 'prefix' after 159 ms
[decoverri@DecoUltra ~/Desktop/app] master! $
```

Mas note que sempre queremos chamar a task `prefix`. Então podemos configurar uma task `default` que vai chamá-la usando a função `start`.

```
gulp.task("default", function(){
  gulp.start("prefix");
});
```

Para chamar a task `default` é só usar o comando `$ gulp`

30.9 EXERCÍCIO: MELHORANDO NOSSAS TASKS

1. **Mude** nossas tasks para gerar uma dependência entre elas. A "prefix" deve depender do "copy".

```
gulp.task("prefix", ["copy"], function(){
```

```
//...
});
```

Agora rode o comando `gulp prefix` e veja o que acontece.

2. Para facilitar ainda mais nossa vida, crie um task default que chama a task "prefix".

```
gulp.task("default", function(){
  gulp.start("prefix");
});
```

3. Quando adicionamos um arquivo, a task **copy** se encarrega de colocar eles na pasta *dist*. Mas e uma alteração? E uma remoção? Queremos que a pasta *dist* represente exatamente o estado da *src*.

Crie uma nova task, para limpar a pasta *dist* sempre antes de copiar. Pra isso, instale o `del` do Node.js pelo terminal.

```
$ npm install del
```

4. Agora, no `gulpfile.js`, use o `require` para pegar o `del` e criar a task. Não esqueça de acrescentar essa dependência no "copy"

```
var del = require("del");

gulp.task("clean", function(cb){
  del(["dist/**/*"], cb);
});

gulp.task("copy", ["clean"], function(){
  //...
});
```

Rode de novo o comando `$ gulp .`

30.10 GULP WATCH

Automatizamos já muita coisa mas ainda precisamos lembrar de rodar o `gulp` na linha de comando. Pior, rodar a cada mudança que fizermos no nosso código. Não parece muito prático.

Podemos usar o plugin `gulp-watch` para executar as tasks pra gente sempre que houver alguma mudança no nosso projeto. Podemos instalá-lo com:

```
npm install gulp-watch
```

Precisamos importar o módulo no `gulpfile`:

```
var watch = require("gulp-watch");
```

Depois disso, há várias formas de usar. Uma delas é tratar o `watch` como um filtro nas tasks que já temos. Por exemplo, na que lida com os prefixos:

```
gulp.task("prefix", function() {
  return gulp.src("src/css/*.css")
```

```
.pipe(watch("src/css/*.css"))
.pipe(autoprefixer())
.pipe(gulp.dest("dist/css/"));
});
```

30.11 EXERCÍCIO: AUTOMATIZANDO A AUTOMATIZAÇÃO COM WATCH

1. Comece instalando o **gulp-watch**. No terminal, faça:

```
$ npm install --save-dev gulp-watch
```

2. Volte ao **gulpfile.js**, Acrescente o `require` necessário

```
var watch = require("gulp-watch");
```

3. Agora crie a task para assistir a modificações dos arquivos do nosso projeto. Faça-o chamar a task "prefix" quando isso ocorrer

```
gulp.task("watch", function(){
  watch("src/**/*", function(){
    gulp.start("prefix");
  });
});
```

4. No terminal, inicie o watch com o comando `gulp watch`. Se o terminal travar, é porque está funcionando. O Watch é uma task que vai ficar rodando enquanto desenvolvemos.

Abra no navegador o **dist/principal.html**, pois ele é que usa os CSSs modificados pelo gulp. Depois faça uma alteração no projeto original. Recarregue o **dist/principal.html** página e veja que a alterão já entrou na pasta dist também.

Veja no terminal as tarefas que o gulp executou sem você fazer nada, apenas por ter feito uma modificação.

APÊNDICE - DESCOMPLICANDO O CSS COM SASS

31.1 PRÉ-PROCESSADORES CSS

O CSS evoluiu muito nos últimos anos. É capaz de fazer coisas fantásticas e antes inimagináveis. Mas como desenvolvedores somos insaciáveis. Sempre queremos algo novo. Alguma funcionalidade extra que ainda não tem no CSS mas que nos ajudaria muito em alguma parte do projeto.

Preenchendo essa lacuna começaram a surgir os pré-processadores CSS. Novas linguagens com recursos mais avançados mas que no fim devem ser compiladas para um CSS comum, que o browser entende.

Existem várias. SASS, LESS, Stylus e outras. Cada uma com suas particularidades e com sua legião de seguidores. Vamos ver um pouco do SASS, que é o mais usado no mercado e um dos mais poderosos.

31.2 SASS

O SASS é bem antigo e já passou por muita coisa. Originalmente escrito em Ruby, já foi implementado em JavaScript e C++. Tem duas variações de sintaxes, uma totalmente inovadora e outra que funciona como uma extensão da sintaxe do CSS. Vamos preferir essa última, a SCSS.

Nesting de seletores

Um recurso bastante comum nos pré-processadores é facilitar a escrita de seletores de vários níveis. Imagine um CSS comum assim:

```
header {  
    width: 90%;  
}  
  
header h1 {  
    font-size: 1.5em;  
}
```

Colocamos certa regra no header e outra no h1 dentro do header. Mas e se mudarmos o `header` para usar uma classe `.cabecalho`? Temos 2 lugares pra mexer. Fora que existe uma certa desorganização no código original: o seletor do h1 está ligado ao do header mas nada impede que

escrevamos tudo espalhado no arquivo.

Existe no SASS então o recurso de escrever seletores aninhados:

```
header {  
    width: 90%;  
  
    h1 {  
        font-size: 1.5em;  
    }  
}
```

Repare que o `h1` é declarado dentro do `header`. Isso não é CSS válido. No fim, o SASS vai gerar um CSS parecido com o que vimos antes, com o seletor `header h1`.

Nesting de media queries

Parecido com a ideia de seletores aninhados, existe um recurso semelhante para media queries. Da forma como funcionam no CSS, é fácil ter que repetir seletores dentro das media queries. Por exemplo:

```
header h1 {  
    font-size: 1.5em;  
}  
  
@media (min-width: 600px) {  
    header h1 {  
        font-size: 2em;  
    }  
}
```

Repare como precisamos escrever o seletor `header h1` duas vezes. Se um dia precisar mudar, há 2 lugares pra fazer isso.

Com SASS, podemos escrever media queries sem seletores dentro de outro seletor. Desta forma:

```
header h1 {  
    font-size: 1.5em;  
  
    @media (min-width: 600px) {  
        font-size: 2em;  
    }  
}
```

Isso não é CSS válido. É um atalho do SASS que permite não duplicar os seletores nas media queries. No fim, será gerado o CSS anterior, com o seletor duplicado.

31.3 COMPILANDO SASS COM GULP

Vimos que o SASS não é uma tecnologia entendida pelo navegador. Para funcionar, precisamos compilar essa linguagem em um arquivo CSS válido. Um cara que consegue fazer isso é o Gulp, e basta apenas adicionar um plugin:

```
npm install gulp-sass
```

E no gulpfile, importar o novo plugin:

```
var sass = require('gulp-sass');
```

Aí podemos escrever uma task nova, bem parecida com as que já fizemos. A diferença é que vamos chamar `sass()` para fazer a compilação:

```
gulp.task('sass', function(){
  return gulp.src('dist/scss/*.scss').
    .pipe(sass())
    .pipe(gulp.dest('dist/css'));
});
```

Mas e se acontecer algum erro na compilação? Podemos pedir para o Gulp imprimir esses erros e assim descobrir onde precisamos corrigir. Basta adicionar o comando dos gulp-sass

```
sass().on('error', sass.logError)
```

```
gulp.task('sass', function(){
  return gulp.src('dist/scss/*.scss').
    .pipe(sass().on('error', sass.logError))
    .pipe(gulp.dest('dist/css'));
});
```

31.4 EXERCÍCIOS: NESTING E SASS COM GULP

1. Vamos facilitar um dos nossos css com o SASS. Crie uma pasta `scss` dentro da pasta do projeto.
2. Dentro dessa nova pasta, crie o arquivo `estilos.scss`. copie para ele todo o conteúdo do `estilos.css` que já existe. Depois mude o novo arquivo para fazer o `nesting` com a media query do `#mudaLayout` :

```
//...
#mudaLayout {
  display: none;

  @media (min-width: 610px){
    display: inline-block;
  }
}
```

3. Agora vamos configurar o `gulp` para ele poder processar o scss. No terminal, use o npm para instalar o `gulp-sass` no projeto:

```
$ npm install --save-dev gulp-sass
```

4. Agora vá ao `gulpfile.js`. Crie uma variável para guardar o `gulp-sass`.

```
var sass = require('gulp-sass');
```

5. Crie uma task que pegue todos os arquivos na pasta `scss`, processe o SASS e salve o resultado na pasta `css`. Vamos fazer tudo isso na pasta `dist`, para não alterar o nosso projeto original:

```

gulp.task('sass', function(){
    return gulp.src('dist/scss/*.scss')
        .pipe(sass())
        .pipe(gulp.dest('dist/css'));
});

```

6. Acrescente o comportamento de mostrar o erro caso não consiga processar o css:

```

gulp.task('sass', function(){
    return gulp.src('dist/scss/*.scss')
        .pipe(sass().on('error', sass.logError))
        .pipe(gulp.dest('dist/css'));
});

```

7. No terminal, rode a task que criamos do gulp:

```
$ gulp sass
```

Abra o arquivo **dist/css/estilos.css** e veja o resultado.

8. (Opcional) Mude as configurações do **gulpfile.js** para que a task de processamento do sass faça parte das dependências das outras tasks. Que ordem você faria as dependências?

31.5 REAPROVEITAMENTO COM MIXINS

É muito comum no CSS precisarmos escrever a mesma coisa em várias partes do código.

Um exemplo: fizemos antes algumas classes modificadores para cartões que mudam o tamanho. Eram a `.cartao--textoPequeno` , `.cartao--textoMedio` e `.cartao--textoGrande` . Além da mudança óbvia do `font-size` , essas classes também ajustavam a largura do cartão e a forma como o flexbox escalonava-os (com `flex-basis`).

Se lembrar do CSS, era algo assim:

```

.cartao--textoPequeno {
    font-size: 1em;
    width: 11em;
    flex-basis: 11em;
}

.cartao--textoMedio {
    font-size: 1.5em;
    width: 9em;
    flex-basis: 9em;
}

.cartao--textoGrande {
    font-size: 2em;
    width: 6em;
    flex-basis: 6em;
}

```

Repare que os valores mudam mas há muita repetição ainda. No SASS, podemos isolar código repetido em **mixins**. É como se fossem funções como as que temos em JavaScript. Recebem parâmetro e

permitem encapsular código nelas.

A sintaxe SASS determina um bloco `@mixin` que recebe um nome e uma lista de parâmetros (variáveis):

```
@mixin tamanhoCartao($fonte, $largura){  
    font-size: $fonte;  
    width: $largura;  
    flex-basis: $largura;  
}
```

E aí podemos usar esse mixing depois com `@include` e passar os valores:

```
.cartao--textoPequeno {  
    @include tamanhoCartao(1em, 11em);  
}  
  
.cartao--textoMedio {  
    @include tamanhoCartao(1.5em, 9em);  
}  
  
.cartao--textoGrande {  
    @include tamanhoCartao(2em, 6em);  
}
```

31.6 EXERCÍCIOS: ISOLANDO CÓDIGO EM MIXINS

1. Na pasta `scss` crie um arquivo `cartao.scss` e copie o código que já existe do arquivo `catao.css`.
2. Crie um mixin antes do css dos diferentes tamanhos de cartão:

```
@mixin tamanhoCartao($fonte, $largura){  
    font-size: $fonte;  
    width: $largura;  
    flex-basis: $largura;  
}
```

3. Agora, mude os tamanhos para usar o mixin que criamos, passando os valores certos de cada um:

```
.cartao--textoPequeno {  
    @include tamanhoCartao(1em, 11em);  
}  
  
.cartao--textoMedio {  
    @include tamanhoCartao(1.5em, 9em);  
}  
  
.cartao--textoGrande {  
    @include tamanhoCartao(2em, 6em);  
}
```

4. Agora rode o gulp novamente no terminal

```
$ gulp sass
```

Procure o arquivo `.css` gerado e veja o resultado. Note que o resultado final também tem prefixos.

Você pode descobrir mais funcionalidades do SASS na documentação do site oficial <http://sass-lang.com/>

31.7 DISCUSSÃO EM AULA: PRECISO MESMO DE UM PRÉ-PROCESSADOR? QUANDO O CSS É O SUFICIENTE?