

Performance Web II: Critical Path, HTTP/2 e Resource Hints: Aula 2

Usando o async

O *javascript* possui, naturalmente, uma característica sequencial e o navegador tenta otimizar algumas coisas, como por exemplo, fazer o download dos quatro *scripts* em paralelo, mas na hora de executar, por padrão, isso é feito na sequência. Um *script* mais lento pode atrasar a execução de todo o resto da página e isso pode virar uma bola de neve. Por isso que no "html5" entrou um novo recurso, o *atributo* `async`. Isso sinaliza ao navegador que determinado *script* pode ser baixado e executado de forma assíncrona, que significa de forma não bloqueante. Um *script* não irá mais ser o motivo de bloqueio da execução do "html" e de outros *scripts*.

Assim, nada nos impede que usemos o `async` em grande parte dos *scripts*, esse seria o cenário ideal:

```
896
897 <script async src="assets/js/menu.js"></script>
898 <script async src="assets/js/busca.js"></script>
899 <script async src="assets/js/detect.js"></script>
900 <script async src="assets/js/footer.js"></script>
901
```

Isso, entretanto, pode gerar um problema: esses *scripts* podem ser executados fora de ordem. Existem alguns *frameworks* no mercado que fazem o carregamento assíncrono com definição de prioridades. Não trataremos disso no curso pois esse é um dos tópicos de "Módulo js". O que estamos dizendo é que se você utiliza algum dos *frameworks* de módulo em *javascript*, você ganha a possibilidade de estabelecer dependência entre *scripts*. No caso desse site isso não é interessante pois ele já foi feito de maneira a que cada script seja auto-contido e que eles sejam independentes entre si e, portanto, que sejam carregados de maneira assíncrona sem problemas.

Agora que temos o *javascript* não mais bloqueante, ele não precisa estar no final da página. Ele pode estar no *head* outra vez. Colocar no *head*, novamente, altera a prioridade de download desses *scripts*. Aliás você pode priorizar, também, os *scripts*, colocando alguns no final e alguns no início. Lembrando que se você jogar o `async` para cima o código pode ser executado assim que ele chegar, não trava a renderização, então teremos que colocar, novamente, o *onload* novamente.

Para fechar essa aula, nós estávamos fazendo a concatenação do *javascript* com o *plugin* do *gulp*. Aquele *plugin* do *gulp* também recebe um padrão adicional que é o parâmetro `async`. O que ele faz é pegar todos os *scripts*, "buildar" em um só na ordem certa e colocar o `async` em um todo. E isso pode ser interessante pois usando o recurso de concatenação é possível juntar *scripts* que tenham dependência e carregar assincronamente da mesma forma.

A dica é utilizar o `async`. Mesmo que ele seja do "html5" e tenhamos navegadores antigos, isso não influencia em praticamente nada, se eles forem MUITO ANTIGOS o que acontecerá é que eles continuarão fazendo o que já faziam antes, carregar na ordem e de maneira bloqueante.

O *javascript* e todos os eventos de usuário são executados em uma única *thread*, uma *single thread*. Apesar de interessante isso indica um gargalo de performance, pois todos os códigos *javascript* estão sendo executados na mesma *threads*, ou seja, estão competindo pelo mesmo processador que as próprias interações do usuário.

Quando acessamos um site e clicamos em algo e ele tranca, isso é um resultado do navegador tentando executar algo e o usuário também. Mesmo que acrescentemos o `async`, ele irá ser baixado, em paralelo e conforme isso acontecer ele vai ser executado:

```
<!-- build:js assets/js/scripts.js async -->
<script async src="assets/js/menu.js"></script>
<script async src="assets/js/busca.js"></script>
<script async src="assets/js/detect.js"></script>
<script async src="assets/js/footer.js"></script>
<!-- endbuild -->
```

E se esse script fizer algo pesado ele vai travando a *thread* que responde

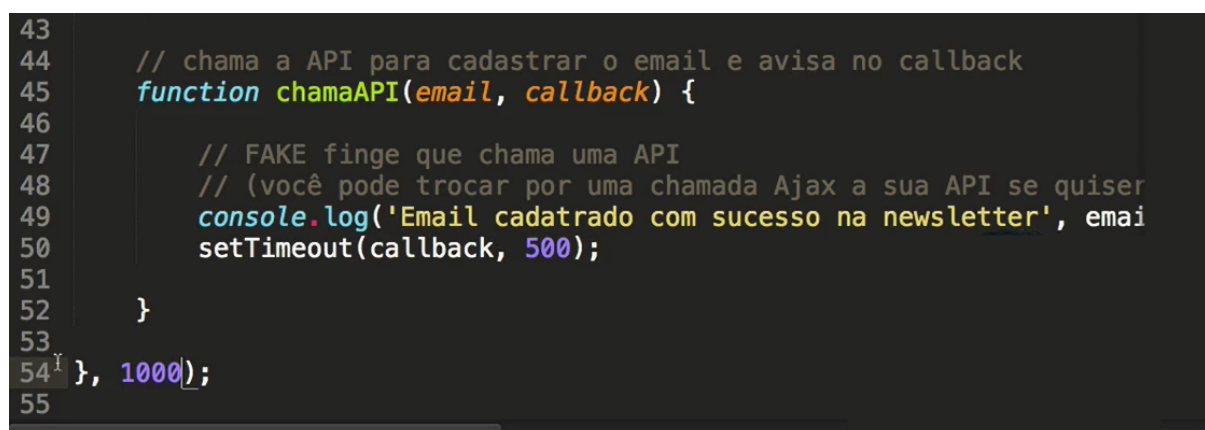
scripts que não são tão essenciais para a página e adiar a execução deles. Por exemplo, temos um determinado *script* que carrega elementos no rodapé da página, o "footer.js". No momento em que acessamos o site do **Alura** queremos fazer uma busca, mas o campo de busca é travado devido a execução do "footer.js" que é responsável pelo rodapé. O problema é que o campo de busca e o rodapé estão competindo entre si.

Podemos adiar a execução do "footer.js" usando um truque, o `setTimeout`. É uma função do *javascript* que aceita receber duas coisas: uma função com o que queremos executar e quanto tempo depois essa função deve ser executada. Assim, você pode dizer que gostaria que a função fosse executada em 1000 milissegundos, `setTimeout(funcao, 1000)`. Estamos, dessa maneira, adiando a execução do código.

O que faremos é passar a função onde o primeiro parâmetro fica no início, como podemos observar na primeira imagem abaixo e depois, passamos o segundo parâmetro, o tempo que deve ir ao final de tudo, como representando na segunda imagem. Observe:



```
1  setTimeout(funcao, 1000);
2
3
4  setTimeout(function() {
5
6      var newsletterButton = document.querySelector('.footer-newsletter
7      var inputEmail = document.querySelector('.footer-newsletter-input
8
9      if (newsletterButton) {
10         newsletterButton.onclick = cadastraNewsletter
11     }
12
13
```



```
43
44     // chama a API para cadastrar o email e avisa no callback
45     function chamaAPI(email, callback) {
46
47         // FAKE finge que chama uma API
48         // (você pode trocar por uma chamada Ajax a sua API se quiser
49         console.log('Email cadastrado com sucesso na newsletter', email
50         setTimeout(callback, 500);
51
52     }
53
54     }, 1000);
55
```

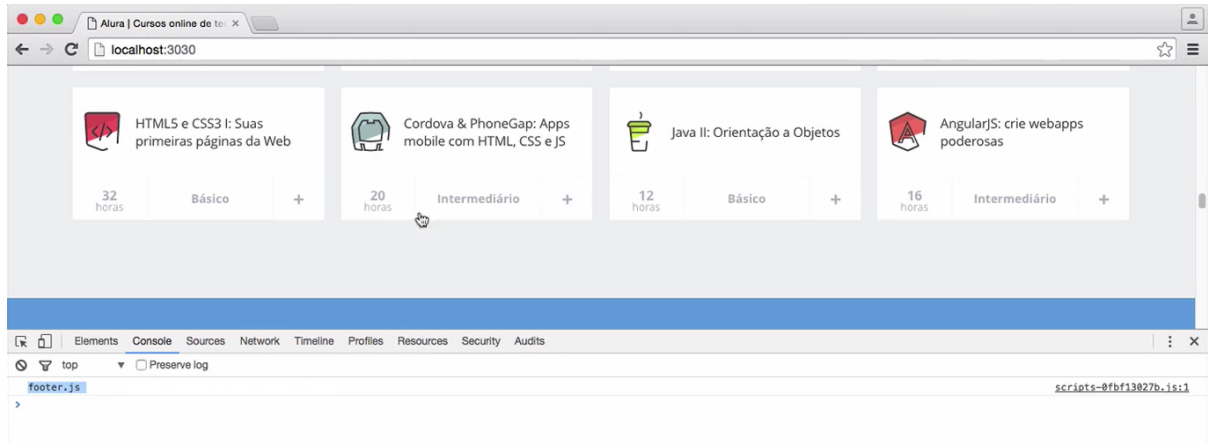
Vamos carregar o "footer.js" de maneira assíncrona só que a execução dele só vai acontecer dentro de 1 segundo. Ou seja, podemos quebrar a execução do código por partes. Se você preferir, pode representar

`console log`, logo abaixo do `setTimeout`, para conferir qual foi o tempo que tardou em executar:

```
console.log('footer.js');
```

Copiar código

Você pode visualizar, através disso, no próprio *console* quando o `footer.js` será carregado.



O que fizemos é tirar *scripts* secundários do processo de execução do navegador para não competir com a *single thread* e outras ações do usuário. Se você desejar pode fazer isso com outros *scripts*.

Lembrando que sempre que modificarmos algo, precisamos dar um `gulp default`.

Vimos o `setTimeout` que serve para adiar a execução, um dos recursos mais eficiente para isso. Com ele, podemos quebrar as várias tarefas que temos em blocos para que se consiga respirar e liberar a *thread* de outras funções do usuário. Não veremos no curso, mas você pode utilizar o `requestAnimationFrame` para adiar animações e pode ser utilizado também o `requestIdleCallback` que permite que determinado código seja executado apenas quando o navegador estiver *Idle*, ou seja, ocioso. Esse último recurso, entretanto, só pode ser usado no *Chrome*.