

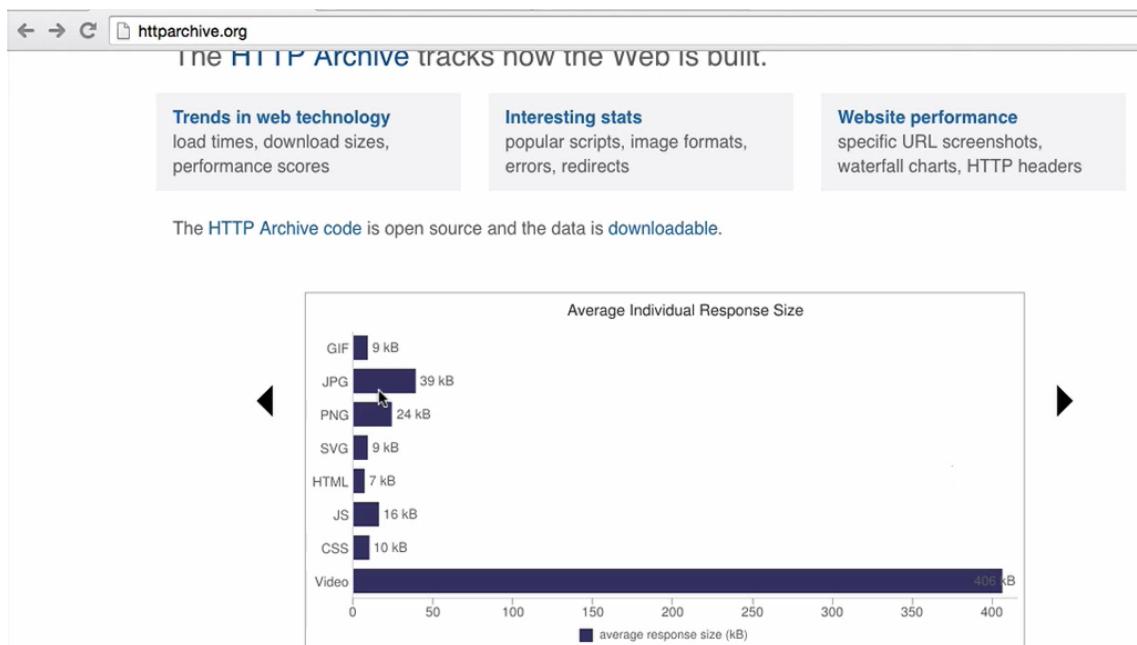
Performance Web II: Critical Path, HTTP/2 e Resource Hints: Aula 5 - Atividade 5

Critical Path 2 | Alura

15-20 minutes

Ferramentas

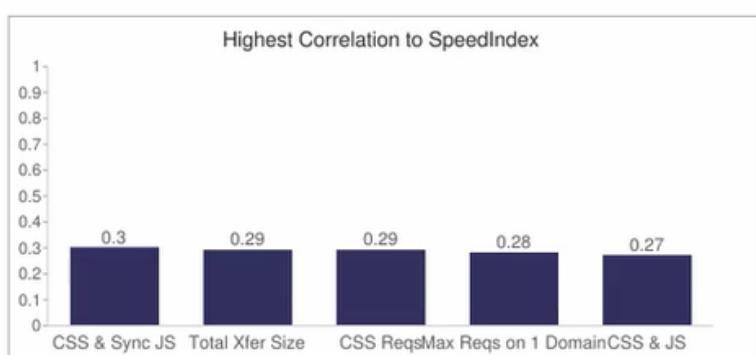
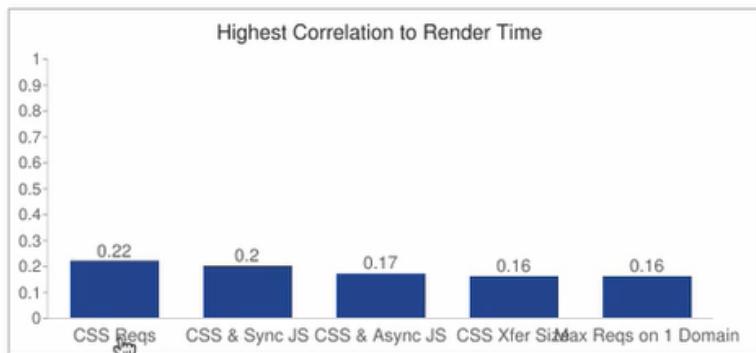
Uma ferramenta bastante interessante para comparar nosso site com outros é o link httparchive.org. Nele podemos verificar diversas informações e gráficos dos sites mais acessados da internet.



Esse gráfico mostra, por exemplo, o tamanho médio de resposta de cada um dos tipos de arquivos descritos. Logo de cara temos os gráficos mais recentes publicados, mas podemos visualizar outros dados e informações. Se clicarmos no item estatísticas teremos diversos gráficos que nos dão as mais diversas informações.

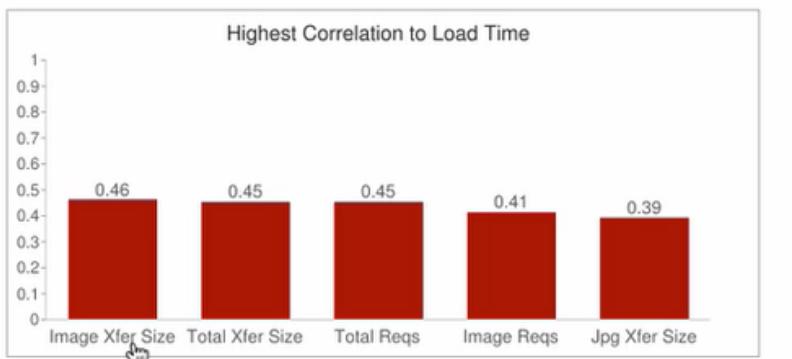
Um outro ponto bastante interessante é se acessarmos a aba *stats* podemos verificar alguns gráficos que tentam encontrar a correlação

entre tempo de carregamento, renderização e alguma prática da página. A ideia é trazer os sites mais rápidos e buscar entender porque eles possuem esse diferencial. Vamos analisar o seguinte gráfico:



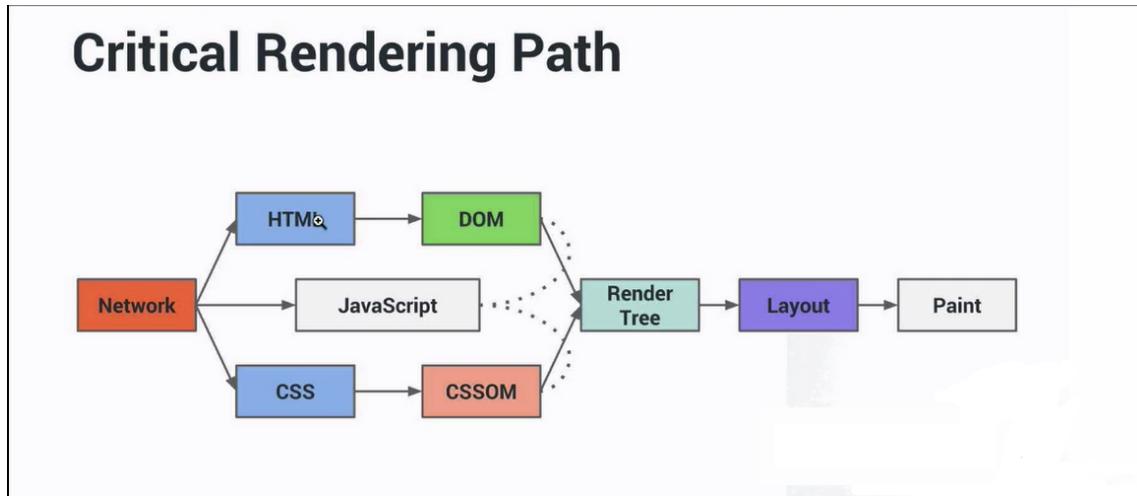
A maior correlação entre o tempo menor de renderização são os *requests* do CSS. Um número maior de CSS afeta a piora da renderização e um número menor melhora. O interessante é perceber que o tempo de renderização e o *Speedindex* são afetados pela forma como o CSS é carregado.

Nesse trecho temos também um gráfico que mostra o tempo total de carregamento de uma página, onde as imagens são o que mais interferem:



Enfim, fique a vontade para explorar essa ferramenta.

Já falamos sobre o *Critical Rendering Path*, vamos retornar rapidamente ao tema. Já falamos que existe uma dependência entre o *Paint* final e os processos anteriores. Observe novamente a imagem:



Não é apenas o *javascript* que bloqueia a renderização da página, é também o CSS. O *javascript* carregando de maneira assíncrona deixa de bloquear a renderização, mas com o CSS isso não ocorre, pois, por natureza ele bloqueia a renderização e não adianta renderizar um html sem estilo. Não é uma boa prática de usabilidade.

Temos um *request* para o "Html" e um distinto para o CSS, isso devido a concatenação que realizamos. Isso gera um problema, pois só conseguiremos chegar no *Paint* depois que esses dois *requests* forem feitos, parcelados, executados e todos os demais passos do processo. Ou seja, para finalizar dependemos de duas requisições.

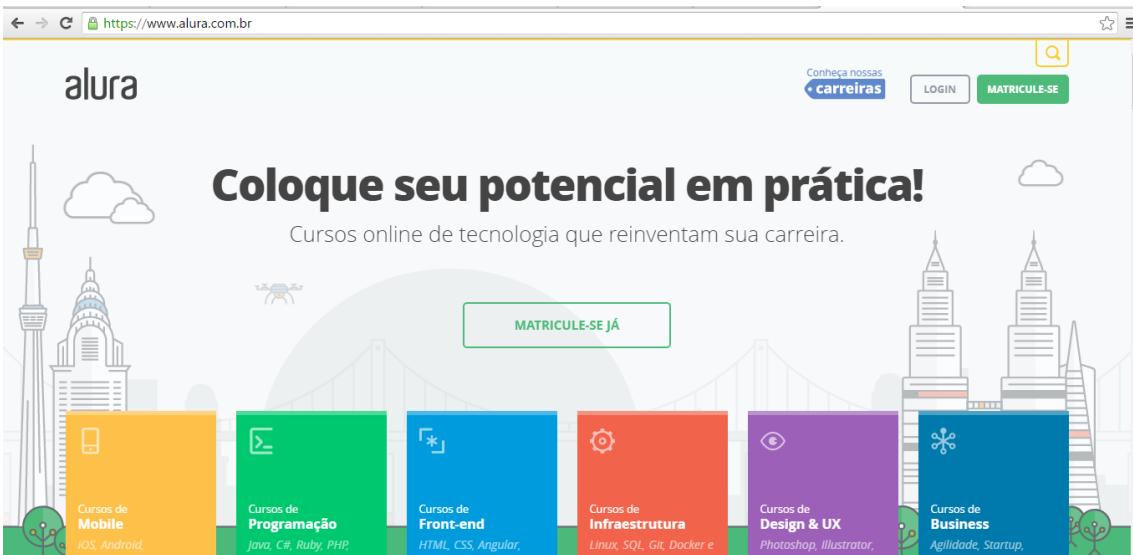
Vamos lembrar nossa conversa anterior do funcionamento da rede. Quando falamos em fazer dois *requests* também estamos falando em fazer dois *RTTs* - *Round Trip time*, ou seja, estaremos pagando duas vezes a latência.

O segredo para destravar o Critical Rendering Path é ter:

Apenas 1 RTT render

Isso significa que, em um cenário ideal, para obtermos ultra-performance é necessário que consigamos com apenas um RTT possuirmos uma renderização significativa para o usuário. Não significa que com apenas um RTT tenhamos que trazer o site inteiro,

isso é impossível, o que devemos fazer é que nesse RTT foquemos em renderizar algo que seja útil para o usuário. Por exemplo, a primeira coisa que vemos quando acessamos o site é o seu topo:



Se priorizarmos a renderização do topo do site e que essa imagem seja visível no momento em que o usuário entrar, teremos uma excelente performance para o usuário. O restante das coisas será carregado em *background*. O que estamos fazendo é levar para um próximo nível o debate sobre *lazy load* e carregamento prioritário.

Os passos são, portanto, os seguintes:

request HTML
Line Inline CSS inicial + Inline Imgs (logo) não precisa de JS / (Inline Js inicial) < 14 KB FLUSH INICIAL

Explicando os passos:

O segredo é fazer em um RTT, apenas, a renderização do topo da página. Quando falamos de 1 RTT estamos falando do *request* que trará a página "HTML", para isso precisamos fazer um *inline* do CSS inicial. Precisamos descobrir qual é o CSS necessário para renderizar pelo menos a parte de cima e usaremos esse CSS *inline*. No mundo ideal não precisamos de *javascript* para a renderização inicial, se tiver, fazemos um *inline* do *javascript* inicial. lembrando que tudo isso deve estar comportado em 14 KB. Basicamente, o truque é trazer no arquivo HTML tudo o que for necessário "inlineado" para a renderização e isso deve ser comportado em 14 KB. Como tudo deverá estar gziptado, cabem bastante coisas. Se você estiver mexendo com o *backend* pode mandar o HTML inicial através do *flush*. Entretanto, não abordaremos

essas técnicas.

Fazemos tudo isso para renderizar o início da página, em relação ao restante, faremos ASYNC. Esse é o grande segredo:

| Trazer inline o início e carregarmos assincronamente o restante.

Existem alguns detalhes importantes, já vimos como fazer *script* assincronamente, agora, CSS já é um pouco mais complicado.

Algumas pessoas se incomodam em fazer inline, isso ocorre pois, por exemplo, o cabeçalho da página é sempre igual e toda vez que acessarmos essa página estaremos baixando a mesma coisa. O *inline*, portanto, acaba se contrapondo ao *cache*. O que estaremos fazendo é um *Tradeoff cache vs Performance Inicial*. Quando abrimos mão do *cache* estamos abrindo mão de economizar bytes pois estaremos baixando a mesma coisa várias vezes.

Temos que pensar se isso realmente valerá a pena e essa é uma decisão pessoal. Em muitos casos vale sim, lembra-se de todas as otimizações que fizemos anteriormente, por causa delas, não vale a pena se dar ao luxo de fazer essa otimização em nome da usabilidade?

Um dos motivos de se usar essa tática é que a performance de renderização fica ótima. Se estivéssemos falando de mais kbytes que além dos 14 que contamos, talvez não valesse a pena. Mas, nesse caso, esse custo talvez vale a pena de enfrentar.

Vamos observar na prática como podemos carregar o HTML priorizando o *Critical Path*. Vamos observar um hipotético arquivo HTML. Temos na página o "estilo.css" que é blocante, enquanto ele estiver no *head* da página ele irá bloquear a renderização dos elementos do *body*. Análise:



```
demo.html
1 <!DOCTYPE html>
2 <html>
3 <head>
4
5   <link rel="stylesheet" href="estilo.css">
6
7 </head>
8 <body>
9
```

A screenshot of a code editor window titled 'demo.html'. The code is displayed in a monospaced font. Line numbers 1 through 9 are shown on the left. The code consists of the standard HTML doctype, opening and closing html, head, and body tags. Inside the head tag, there is a single line of code: a link element with 'rel="stylesheet"' and 'href="estilo.css"'. The editor interface includes a toolbar at the top and a status bar at the bottom indicating the file name and path.

```
10 ...
11 ...
12 ...
13 ...
14 </body>
15 </html>
```

O que desejamos fazer é quebrar o "estilo.css" em dois arquivos diferentes, um que contenha o início da página, o "inicial.css" e outro arquivo que possua o "resto.css". Evitaremos que o CSS seja carregado externamente e iremos carregar ele dentro de um *style*, assim, todo o CSS inicial renderizará o conteúdo do *above the fold* da página, ou seja, isso será feito *inline* no "estilo". Na verdade, o que fizemos ainda não foi de grande ajuda, pois o CSS continua blocante. Precisamos carregar o CSS de maneira assíncrona para resolver esse problema.

Essa tarefa, entretanto, não é fácil. Para fazer isso com o *javascript* a boa prática era colocar ao final da página o código para que nada fosse bloqueado por ele. Porém, no CSS o *browser* não permite essa prática (a maior parte ao menos não permite). O CSS, entretanto, deve ser carregado no *head* e ele é blocante por natureza. No CSS tão pouco existe a opção de carregar de modo ASYNC. Uma solução famosa é usar um *framework* chamado *loadCSS* que faz o carregamento de forma assíncrona. Podemos utilizar o seguinte:

```
<script>loadCSS('resto.css');</script>
```

Mas, essa solução é baseada em *javascript* e ela pode acarretar uma série de preocupações, não é uma solução tão limpa.

Podemos fazer isso de maneira inativa no *browser*, essa opção, entretanto, possui alguns pormenores. A tag acompanhada do "rel" pode ser utilizada para carregar diversas coisas. Existe um "rel" novo a partir das últimas versões dos navegadores que é o "preload". Podemos passar para o "rel=preload" o "resto.css" que ele fará o download do arquivo, mas sem executar o arquivo. O download é feito de maneira assíncrona e coloca o "resto.css" no *cache* do navegador. Essa tag

nova é capaz de carregar qualquer coisa de maneira assíncrona, mas, como mencionado, sem executar essa coisa. Podemos até passar o que queremos carregar:

```
<link rel="preload" as="style" href="resto.css">
```

Com isso, estamos dizendo que queremos fazer o preload como um style. Ele dará a mesma prioridade de download que outros estilos da página, mas apenas o download, ele não executará esse arquivo.

Como o "preload" não bloqueia a renderização, ele não executa esse arquivo podemos usar o atributo "onload" que dispara quando o arquivo termina de carregar (de maneira assíncrona). Agora, vamos fazer um truque simples porém poderoso.

```
<link rel="preload" as="style" href="resto.css"  
onload="this.rel='stylesheet'">
```

Isso significa que ele pegará o atributo "rel" e trocará de "preload" para "stylesheet". Efetivamente, estamos transformando isso em um link "rel="stylesheet" que na prática executa e renderiza o css. Em uma linha fizemos um carregamento assíncrono de CSS, que quando terminar de ser carregado vai fazer a renderização e executar esse arquivo, e fizemos isso usando apenas o "onload". Se você não tiver o *javascript* habilitado existe a opção de utilizar o *fall back*. Você pode usar ele embaixo, junto ao "rel=stylesheet" , que é blocante, mas se o *javascript* não está habilitado essa é a melhor opção:

```
<nonscript><link rel="stylesheet" href="resto.css">  
</nonscript>
```

Com isso já podemos apagar aquilo que tinha sido proposto no início e apresentava diversos problemas. O seguinte:

```
<script src="loadcss.js"></script>  
<script>loadCSS('resto.css');</script>
```

Com isso, conseguiremos carregar o CSS de maneira assíncrona. Essa prática, entretanto, não funciona em todos os navegadores, mas o nosso desejo é que funcione mesmo nos navegadores mais antigos. A solução para lidar com esse problema é usar um *polyfill* para o

helpload. Para saber como utilizar o "loadCSS" você pode buscar informações no *Google*, por exemplo, temos essa maneira apresentada no seguinte site:

The screenshot shows a browser window with the URL <https://github.com/filamentgroup/loadCSS>. The page title is "Recommended Usage Pattern". The content discusses the standard markup pattern for loading CSS asynchronously using the `<link rel='preload'>` and `<link rel='stylesheet'>` combination, along with a `<noscript>` fallback. It includes sample code and notes about browser support and polyfills.

```
<link rel="preload" href="path/to/mystylesheet.css" as="style" onload="this.rel='stylesheet'">
<noscript><link rel="stylesheet" href="path/to/mystylesheet.css"></noscript>
```

Since `rel=preload` does not apply the CSS on its own (it merely fetches it), there is an `onLoad` handler on the `link` that will do that for us as soon as the CSS finishes loading. Since this step requires JavaScript, you may want to include an ordinary reference to your CSS file as well, using a `noscript` element to ensure it only applies in non-JavaScript settings.

With that markup in the `head` of your page, include the `loadCSS` script, as well as the `loadCSS` `rel=preload` polyfill script in your page (inline to run right away, or in an external file if the CSS is low-priority).

No further configuration is needed, as these scripts will automatically detect if the browsers supports `rel=preload`, and if it does not, they will find CSS files referenced in the DOM and preload them using `loadCSS`. In browsers that natively support `rel=preload`, these scripts will do nothing, allowing the browser to load and apply the asynchronous CSS (note the

Como queremos fazer o *polyfill* para os navegadores antigos devemos carregar dois *scripts*, e eles podem ficar no fim da página, de maneira não blocante:

```
<script src="loadcss.js"></script>
<script src="cssrelpreload.js"></script>
```

Carregando isso, resolvemos o problema dos navegadores mais antigos. Se preferir carregar de maneira assíncrona pode ser possível e nesse caso, o código pode ser deslocado, novamente, para o `head`, pois não será mais blocante. O truque para carregar o CSS de maneira assíncrona foi quebrar o CSS em dois, inicial e o resto. O inicial nós colocamos *inline* e o resto nós carregamos de maneira assíncrona usando o "helpload", com um *fall back* e *polyfil* para navegadores antigos. Esse é o princípio básico para o 1 RTT render do *above the fold*. Se você ainda tiver *javascript* necessário, faça *inline*. Já colocaremos isso na prática no projeto. Temos, por fim, o seguinte:

```
demo.html
1 <!DOCTYPE html>
2 <html>
3 <head>
4
5   <!-- <link rel="stylesheet" href="inicial.css"> -->
6   <style>... inicial ...</style>
7
8   <link rel="preload" as="style" href="resto.css" onload="this.rel='stylesheet'">
9   <noscript><link rel="stylesheet" href="resto.css"></noscript>
10
11  <script src="loadcss.js" async></script>
```

```

11   <script src="node_modules/async/dist/async.js" type="text/javascript">
12   <script src="cssrelpreload.js" async></script>
13
14 </head>
15 <body>
16
17 ...
18 ...
19 ...
20
21
22

```

Aplicando Conceitos

A ideia é aplicar agora, todos os conceitos que vimos. O primeiro passo será quebrar o CSS em duas partes, o "inicial" e o "resto". Vamos abrir o "index.html" e nele podemos verificar a quantidade de CSSs sendo "buildados". Temos aqui diversos arquivos que são importantes para o *above the fold*. Vamos separando os arquivos, a mão, que são parte do topo da página. Assim, acabamos de quebrar em duas partes nosso CSS. O "inicial" e o "resto" que está no "estilo.css". Observemos o "estilo.css":

```

24
25   <!-- build:css assets/css/estilos.css -->
26   <link rel="stylesheet" href="assets/css/block-conteudo.css">
27   <link rel="stylesheet" href="assets/css/block-cursoCard.css">
28   <link rel="stylesheet" href="assets/css/block-depoimentos.css">
29   <link rel="stylesheet" href="assets/css/block-elasticMedia.css">
30   <link rel="stylesheet" href="assets/css/block-footer-listaCursos.css">
31   <link rel="stylesheet" href="assets/css/block-footer.css">
32   <link rel="stylesheet" href="assets/css/block-form-erro.css">
33   <link rel="stylesheet" href="assets/css/block-grupoCaelum.css">
34   <link rel="stylesheet" href="assets/css/block-highlighted.css">
35   <link rel="stylesheet" href="assets/css/block-paineisPlanos.css">
36   <link rel="stylesheet" href="assets/css/block-titulos.css">
37   <link rel="stylesheet" href="assets/css/home-aprenda.css">
38
39   <!-- endbuild -->

```

E o "inicial.css":

```

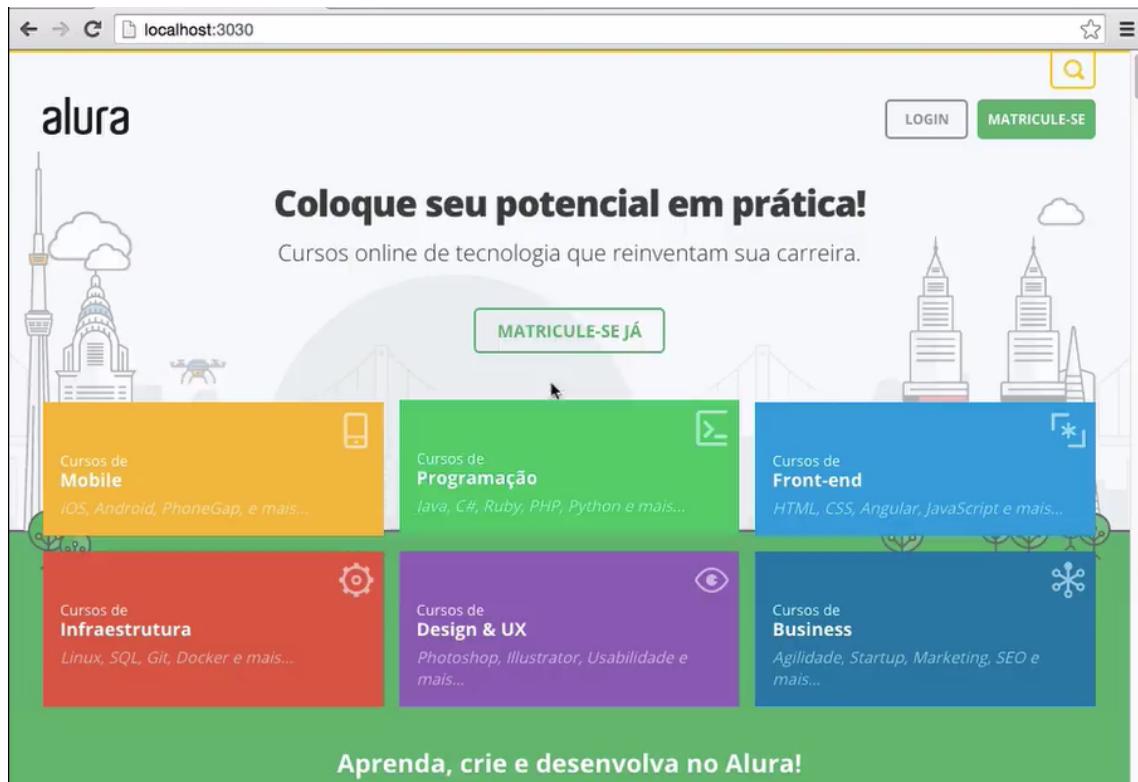
8
9   <link rel="stylesheet" href="assets/css/reset.css">
10  <link rel="stylesheet" href="assets/css/base.css">
11  <link rel="stylesheet" href="assets/css/colors.css">
12  <link rel="stylesheet" href="assets/css/font.css">
13
14  <link rel="stylesheet" href="assets/css/block-header-busca.css">
15  <link rel="stylesheet" href="assets/css/block-header.css">
16  <link rel="stylesheet" href="assets/css/block-buttons.css">
17  <link rel="stylesheet" href="assets/css/block-categoriaCard.css">
18  <link rel="stylesheet" href="assets/css/block-titulo-destaque.css">
19
20  <link rel="stylesheet" href="assets/css/home-diferenciais.css">
21  <link rel="stylesheet" href="assets/css/home-fundo.css">
22  <link rel="stylesheet" href="assets/css/home.css">
23

```

A ideia é que tudo isso seja colocado *inline*. Não iremos ficar copiando e colando a *tag style* em todos esses cerca de dez arquivos, o que podemos fazer é usar o *pluggin* do *gulp* para trabalhar *inline*. Anotaremos a *tag inline* em todos os arquivos e na hora do *build* ele vai gerar tudo *inline*:

```
<link inline rel="stylesheet" href="assets/css/reset.css">
<link inline rel="stylesheet" href="assets/css/base.css">
<link inline rel="stylesheet" href="assets/css/colors.css">
<link inline rel="stylesheet" href="assets/css/font.css">
```

Podemos rodar o gulp no Terminal e ele irá "buildar" tudo na pasta "index.html-dist" de maneira *inline* no "estilo". Podemos salvar isso e rodando o navegador teremos o topo certinho:



Já conseguimos fazer a divisão de maneira que o "inicial" está certo. Só que estamos fazendo isso carregando o "resto.css" de maneira síncrona. Vamos carregá-lo de maneira assíncrona usando o truque do:

```
<link rel="preload" as="style"
      href="assets/css/resto.css"
      onload="this.rel='stylesheet'">
```

Além disso, devemos acrescentar abaixo disso o *fall back* para navegadores sem CSS. E para os navegadores que não suportam o link *rel="preload"* devemos adicionar o *loadCSS* e o *polyfil* que já deixamos no projeto comutado. Acrescentaremos abaixo do *fall back* o seguinte:

```
<script async src="assets/js/loadcss.js"></script>
<script async src="assets/js/cssrelpreload.js"></script>
```

Teremos:

```
39      <link rel="preload" as="style"
40          href="assets/css/resto.css"
41          onload="this.rel='stylesheet'">
42      <noscript><link rel="stylesheet" href="assets/css/resto.css"></noscript>
43      ...
44      <script async src="assets/js/loadcss.js"></script>
45      <script async src="assets/js/cssrelpreload.js"></script>
46
47
```

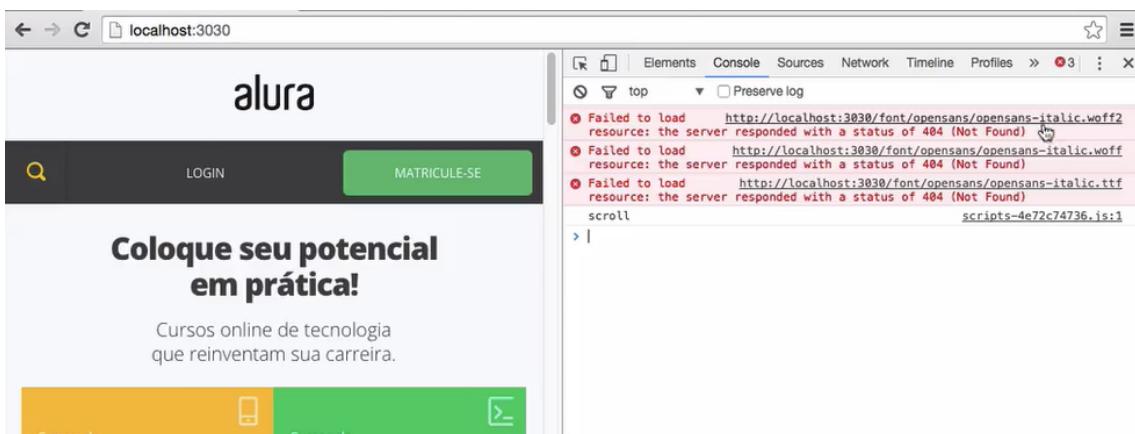
Se "buildarmos" novamente, veremos um detalhe, que na pasta "dist" temos todas as tags que acrescentamos. Só que, o pluggin do gulp ainda está gerando o `<link rel="stylesheet" href="assets/css/resto-6ea5a5cdca.css">` o que é meio problemático. Vamos usar um truque, vamos colocar todos os arquivos que estão no "resto.css" dentro de um comentário para que quando "buildarmos" o que for gerado fique dentro do comentário.

```
24      <!--
25      <!# build:css assets/css/resto.css -->
26      <link rel="stylesheet" href="assets/css/block-conteudo.css">
27      <link rel="stylesheet" href="assets/css/block-cursoCard.css">
28      <link rel="stylesheet" href="assets/css/block-depoimentos.css">
29      <link rel="stylesheet" href="assets/css/block-elasticMedia.css">
30      <link rel="stylesheet" href="assets/css/block-footer-listaCursos.css">
31      <link rel="stylesheet" href="assets/css/block-footer.css">
32      <link rel="stylesheet" href="assets/css/block-form-erro.css">
33      <link rel="stylesheet" href="assets/css/block-grupoCaelum.css">
34      <link rel="stylesheet" href="assets/css/block-highlighted.css">
35      <link rel="stylesheet" href="assets/css/block-painelPlanos.css">
36      <link rel="stylesheet" href="assets/css/block-titulos.css">
37      <link rel="stylesheet" href="assets/css/home-aprenda.css">
38      <!-- endbuild -->
39      -->
```

Assim, o `<link rel="stylesheet" href="assets/css/resto-6ea5a5cdca.css">` será ignorado.

Se testarmos no navegador veremos que estará tudo em ordem.

Vamos retornar e observar alguns detalhes. Vamos no navegador e na aba *network* do console veremos que as fontes vão estar acusando *not found*.





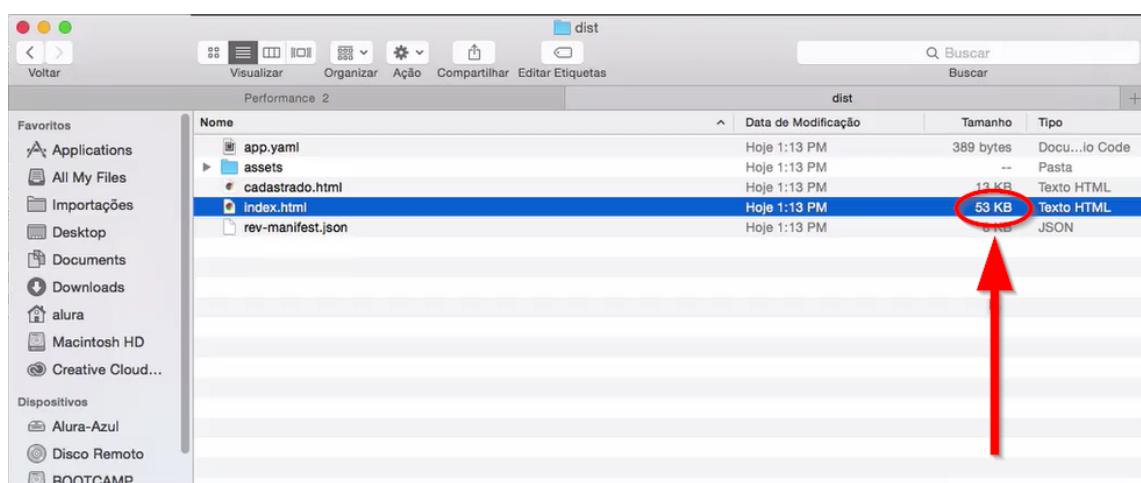
Isso acontece porque trocamos o CSS de lugar, movemos o "resto.css" para o *inline*. Precisamos tomar cuidado com os caminhos das coisas. Agora, temos apenas que ajeitar as fontes para que elas sejam referentes a pasta "assets" pois, elas devem passar a ser relativas ao "index.html":

```
1 @font-face {  
2     font-family: 'Open Sans';  
3     src: local('Open Sans Light'), local('OpenSans-Ligh  
4         url('assets/font/opensans/opensans-light.woff2  
5         url('assets/font/opensans/opensans-light.woff'  
6         url('assets/font/opensans/opensans-light.ttf')  
7     font-style: normal;  
8     font-weight: 300;  
9 }  
10  
11 @font-face {  
12     font-family: 'Open Sans';  
13     src:  
14         local('Open Sans'), local('OpenSans'),  
15         url('../font/opensans/opensans-regular.woff2') for  
16             url('../font/opensans/opensans-regular.woff')  
17             url('../font/opensans/opensans-regular.ttf') f  
18     font-style: normal;  
19     font-weight: 400;  
20 }
```

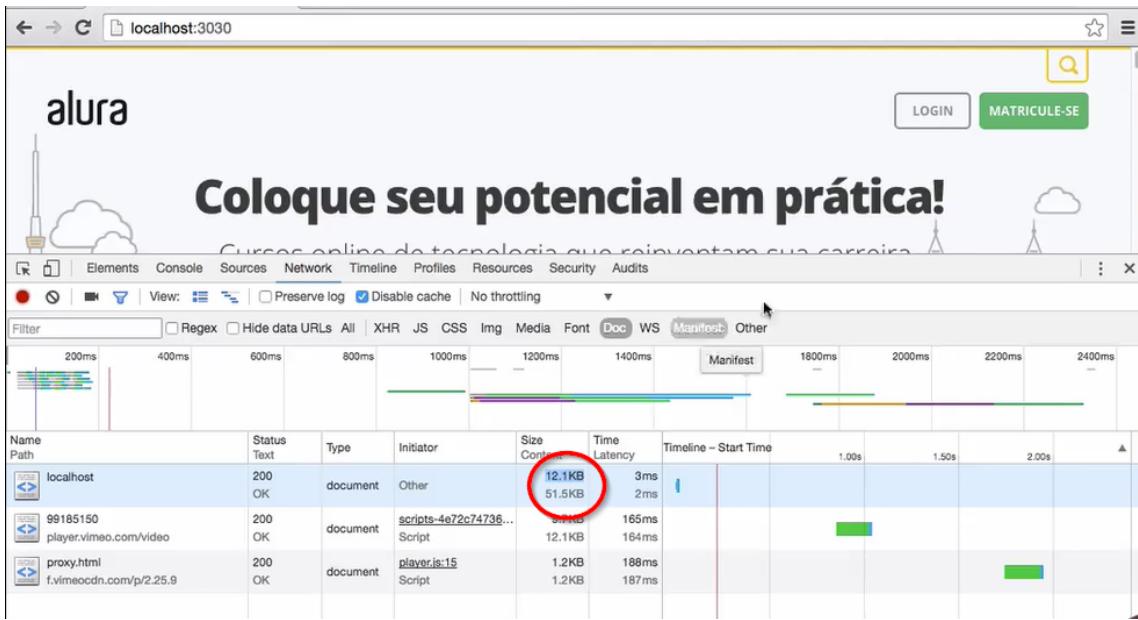
Find What: `../font`

Replace With: `assets/font`

Se dermos um gulp e observarmos mais uma vez o console as fontes estarão sendo carregadas normalmente. Esses errinhos vão ser consertados aos poucos, eles são frutos das modificações que fizemos. Vamos verificar se conseguimos que o index.html tenha ficado com 14 KB:



Na verdade, aparentemente o número ficou muito aquém daquilo que esperávamos, temos que observar o tamanho do "index.html" em um formato gzipado. Se abrirmos o console na aba "Network > doc", não esquecendo que temos que selecionar a opção "*disable cache*" veremos que o arquivo está, na verdade, com apenas 12 KB. Nossa objetivo foi alcançado, conseguimos encaixar todo o nosso CSS inicial.



Passamos por todos esses passos para que você consiga realizar a renderização inicial em apenas 1 RTT, é possível utilizar essa prática em diversos cenários, mas, muitas vezes o cenário é mais complicado. Se não é possível encaixar em apenas 1 RTT, a segunda melhor opção é baixar em 2 RTT ou mais. Mas, tendo em mente que a ideia não é passar ao download/renderização inicial milhares de arquivos que bloqueiam o carregamento e que podem fazer com que o *Critical Render Path* fique travado. Priorizar a renderização inicial melhora e muito a performance.

Uma pergunta que pode ter ficado é como saber quais os CSS que desejamos que sejam escritos *inline*. Essa parte não é simples, se você foi a pessoa que escreveu o próprio CSS é bastante provável que saiba como eles devem ser separados. Frequentemente, se automatiza esse processo, uma vez que é mais fácil pois teremos que repeti-lo em diversas páginas.

Existe um *frame* chamado *Critical*, disponível em <https://github.com>

[/addyosmani/critical](https://github.com/addyosmani/critical). Esse *frame* é responsável por extrair e fazer um *inline* do "CSS" crítico da página e ele pode ser chamado de diversas maneiras na linha de comando, uma delas é a seguinte:

The screenshot shows the GitHub repository page for the `critical` package. At the top, there's a table with configuration options: `include` (array, []), `ignore` (array, []), and `ignoreOptions` (object, {}). Below the table, there's a section titled **CLI** with examples of how to use the command-line interface. It includes examples for standard input and passing a critical CSS file as an option. There's also a section titled **Gulp** with a snippet of Gulp.js code demonstrating how to use the `critical` stream.

			path to start with / or it wouldn't work.)
include	array	[]	Force include css rules. See penthouse#usage .
ignore	array	[]	Ignore css rules. See filter-css for usage examples.
ignoreOptions	object	{}	Ignore options. See filter-css#options .

CLI

```
$ npm install -g critical
```

critical works well with standard input.

```
$ cat test/fixtures/index.html | critical --base test/fixtures --inline > index.critical.html
```

You can also pass in the critical CSS file as an option.

```
$ critical test/fixtures/index.html --base test/fixtures > critical.css
```

Gulp

```
var critical = require('critical').stream;
// Generate & Inline Critical-path CSS
gulp.task('critical', function () {
```

Ele não é perfeito, mas é um bom ponto de partida.