Execute npm install -g gulp-cli para instalar o Gulp globalmente.

Instalar o servidor nginx e habilitar o gzip. Ele serve para o servidor compctar todos os arquivos ao enviar e depois o navegador deszipa.

Instalar o servidor no windows

1 - baixar o zip e descompactar no c:

2 - Prompt: C:\nginx> start .\nginx.exe

Para ver os processos do nginx rodando no momento

1 - Prompt: C:\nginx> tasklist /fi "imagename eq nginx.exe"

Depois disso pode acessar o localhost que ira abrir uma pagina de boas vindas

Para acessar uma pasta de alguma projeto é necessário alterar o arquivo de configuração nginx apontando o caminho do projeto

Por padrão o gzip compacta apenas arquivos de texto, para que ele tbm faça isso para css, js etc é preciso da configuração gzip\_types

server {

listen 80;

server\_name localhost;

gzip on;

gzip\_types text/css application/javascript image/svg+xml;

location / {

#root html;

root C:/Users/lsouza/Desktop/Fontes/Alura/PerformaceWebI-otimizandoofront-end/site";

index index.html index.htm;

}

}

Falamos bastante do GZIP e sua importância. Mas há outras alternativas para compressão do conteúdo trafegado entre cliente e servidor. O GZIP é o clássico que funciona em todos os browsers e servidores, mas há algoritmos mais modernos.

O **Brotli** em especial é um que tem ganho bastante atenção ultimamente. Ele foi criado pelo Google e [anunciado em Setembro de 2015](https://google-opensource.blogspot.com.br/2015/09/introducing-brotli-new-compression.html). Segundo estudos, ele tem uma taxa de compressão 25% melhor que o GZIP e com uma velocidade de processamento parecida.

Ele já possui suporte no Firefox, Chrome e Opera há algum tempo. O detalhe é que funciona apenas em conexões HTTPS (pra evitar quebrar intermediários não compatíveis). Nos servidores, há módulos opcionais para vários deles, como nginx e Apache.

Otimização de imagens é muito importante ja que que segundo estatisticas do site httparchive é o que mais gera tempo de requisiçao por serem grandes.

O site kraken.io ou o tinypng,vc envia suas imagens e remove coisas desnecessárias delas, como metadados(qual o tamanho da abertura da lente, orientação do celular na hora de tirar a foto, data)

Nesse site tem dois modos de otimização, o lossy e o lossless. O primeiro é otimização com perdas e sem perdas respectivamente

Tem o SVGOMG para otimização de svg.

Ferramentas offline

Imageoptim.com - lossless, vantagens dele é que substitui os arquivos originais depois de otimizados automaticamente e joga na lixeira

Existe o riotimage também que faz a mesma coisa, e tem até mais recursos

Ferramentas por linha de comando:

Jpegtran serve para otimizar imagens jpeg

Pngcrush serve para otimizar imagens png

Svgo serve para otimizar imagens svg

Mogrify para otimizar o tamanho correto das imagens

Relembrando algumas das online que mostrei: [kraken](https://kraken.io/web-interface), [tinypng](https://tinypng.com/), [jpegmini](http://www.jpegmini.com/) e [svgomg](https://jakearchibald.github.io/svgomg/).

Também falei de versões Desktop de duas ferramentas famosas: [ImageOptim](https://imageoptim.com/pt-br.html) (Mac) e [RIOT](http://luci.criosweb.ro/riot/) (Windows).

Mas em linha de comando **um mundo de possibilidades se abre**. Algumas que acho bem promissoras:

* [mozjpeg](https://github.com/mozilla/mozjpeg) - Recompressor de JPEG feito pela Mozilla que gera arquivos bem menores e com melhor qualidade, ainda mantendo compatibilidade com o JPEG ([mais aqui](http://calendar.perfplanet.com/2014/mozjpeg-3-0/) e [versão online aqui](https://imageoptim.com/mozjpeg)).
* [pngnq](https://sourceforge.net/projects/pngnqs9/) / [pngquant](https://pngquant.org/) - Mudam um PNG pra PNG8 com palheta fixa de 256 cores. É uma compressão lossy com excelentes resultados em desenhos menos complexos (como ícones). Tem uma GUI excelente chamada [ImageAlpha](https://pngmini.com/) (Mac).

E indo além de apenas otimizar as imagens:

* [ZorroSVG](http://quasimondo.com/ZorroSVG/) - Permite criar "JPEGs com transparência". Muita gente usa foto em PNG por causa da transparência mas isso é **péssimo** pra performance. O zorro usa filtros SVG pra separar a layer de transparência em uma mask separada. Na prática, JPEG com transparência.

Algumas ferramentas de análise de performance exigem que nosso site esteja online em uma URL pública. Eu gosto bastante do Google App Engine por sua simplicidade e por ter um plano free razoável. Então usarei esse serviço aqui para subir uma versão do site.

1) Se não fez ainda, instale o [Google App Engine SDK for PHP](https://cloud.google.com/appengine/downloads#Google_App_Engine_SDK_for_PHP). Adicione o projeto a ele.

2) Precisamos criar um novo projeto no Google Cloud Console. Acesse o [Dashboard](https://console.cloud.google.com/appengine) e vá em **Criar Projeto**. Escolha um nome e ID único. Copie o nome desse ID.

3) No nosso projeto, abra o arquivo app.yaml e adicione a configuração do nome da aplicação que você acabou de criar:

application: id\_do\_seu\_projeto\_aqui

4) Abra o Google App Engine Launcher na sua máquina e clique no botão **Deploy**.

Se tudo der certo você deve conseguir abrir agora a URL:

[http://id\_do\_seu\_projeto.appspot.com](http://id_do_seu_projeto.appspot.com/)

Se preferir, use a versão que fizemos no curso. Deixei uma versão deployada exatamente nesse ponto do projeto em:

[http://wpt-dot-curso-alura-site.appspot.com](http://wpt-dot-curso-alura-site.appspot.com/)

Mostra uma nota para o site em questões de performance

<https://developers.google.com/speed/pagespeed/insights/?hl=pt-BR>

Abra o WebPageTest e rode uma análise de performance no nosso site.

1) Vá em <http://www.webpagetest.org/> e coloque o endereço público do nosso site.

(se preferir, use a minha versão online: [http://wpt-dot-curso-alura-site.appspot.com](http://wpt-dot-curso-alura-site.appspot.com/))

2) Observe algumas métricas como tempo de carregamento, quantidade de requests, tamanho total, as notas que ele dá.

Observe também o gráfico waterfall dos requests. Explore-o, investigue o site. Repare no gráfico de connections como vimos em aula (mostrando as 6 conexões simultâneas).

3) Explore outras possibilidades. Volte a home do WebpageTest e escolha outras opções. Mude a localização e a velocidade de conexão. Mude outras coisas.

Fuce bastante. O WebPageTest é uma ferramenta ***muito*** útil para o profissional de performance front-end.

Vamos fazer a concatenação de todos os CSS em apenas um só, economizando dezenas de requests. Para isso, como vimos em aula, vou usar o **gulp** com **useref**.

1) Edite o arquivo site/index.html e envolva todos os <link> que carregam CSS em dois comentários especiais:

<!-- build:css assets/css/estilos.css -->

<link rel="stylesheet" href="assets/css/reset.css">

....

<!-- endbuild -->

2) No terminal, rode gulp useref

3) Observe o arquivo dist/index.html e o novo arquivo estilos.css

Crie uma sprite CSS com os 3 ícones PNG dos diferenciais do Alura, como vimos em aula. A técnica consiste em criar um PNG único com os 3 desenhos e depois posicioná-los com CSS usando background-position.

## Opção 1: técnica manual

Criar essa sprite manualmente não é algo tão trabalhoso. Podemos gerar a imagem e fazer poucas alterações no CSS.

1) Para juntar os 3 ícones em um único PNG você pode abrir seu editor de imagens favorito e apenas editar as imagens. Ou, se já tiver o [ImageMagick](https://www.imagemagick.org/), criar o arquivo num único comando:

convert site/assets/img/\*.png -append site/assets/img/diferenciais.png

Ele cospe esse arquivo diferenciais.png com os 3 desenhos.

(se quiser, baixe minha versão pronta [aqui](https://raw.githubusercontent.com/alura-cursos/performance-web/773d7e75342fa0d43d1b4d292232372a1f6607f5/dist/assets/img/diferenciais.png))

2) Edite o CSS para usar esse nosso novo arquivo.

a) Na regra .home-diferenciais-icone:before adicione background-image: url("../../assets/img/diferenciais.png");

b) Remova o background-image de cada um dos 3 ícones nas regras seguintes.

c) Em cada um deles, faça um background-position escolhendo um dos ícones. O primeiro é top, o do meio é center e o de baixo, bottom

O diff completo das mudanças:



## Opção 2: ferramentas de geração de sprite

Existem várias. A maior questão é que o CSS gerado não se adapta exatamente ao nosso projeto. Então você vai precisar ajustar ou o CSS gerado ou se markup para se encaixar na ferramenta. Mas vale o teste.

O **sprity** em particular você instala com npm install -g sprity-cli.

E aí pode gerar a imagem e o CSS com:

sprity create /tmp/ assets/img/\*.png -s sprite.css

Abra o sprite.css e observe as regras. Adapate o HTML para usar essa versão.

Como vimos na aula, as sprites SVG são bem mais práticas. Basta criar um novo arquivo e transformar os <svg> em <symbol>, dando um id para cada um.

1) Crie um novo arquivo categorias.svg com um esqueleto básico:

<svg width="0" height="0" version="1.1" xmlns="http://www.w3.org/2000/svg">

<defs>

</defs>

</svg>

2) Dentro do <defs> vamos colocar cada um dos ícones. Basta abrir, por exemplo, o arquivo mobile.svg, copiar seu conteúdo, substituir <svg> por <symbol> e dar um id="mobile" para ele. Algo assim:

<svg width="0" height="0" version="1.1" xmlns="http://www.w3.org/2000/svg">

<defs>

<symbol id="mobile" viewBox="0 0 22 33">

<path d="M17.612 29.584H3.604c-.384 0-.662-.304-.662-.76v-5.51h15.332v5.51c0 .456-.278.76-.662.76zM3.604 2.942h14.008c.384 0 .662.29.662.654v17.396H2.942V3.596c0-.364.278-.654.662-.654zm17.613.697A3.64 3.64 0 0 0 17.577 0H3.64A3.64 3.64 0 0 0 0 3.64v25.24a3.64 3.64 0 0 0 3.64 3.64h13.937a3.64 3.64 0 0 0 3.64-3.64V3.64zM10.837 25h-.175C9.744 25 9 25.783 9 26.75s.744 1.75 1.662 1.75h.175c.92 0 1.663-.783 1.663-1.75S11.756 25 10.837 25"/>

</symbol>

</defs>

</svg>

(Dica: copie a versão do mobile.svg do dist que já foi minificado. Fica mais fácil de enxergar as coisas.)

3) Para usar na página, basta substituir o <img> por um <svg> com <use>:

<svg class="categoriaCard-item-icone"><use xlink:href="assets/img/categorias.svg#mobile"/></svg>

4) Agora faça isso para os 6 ícones das categorias!

Se quiser, tente usar alguma ferramenta automática de geração de SVG como o [svg-sprite](https://github.com/jkphl/svg-sprite).

Use o plugin gulp-inline-source para embutir pequenos arquivos na resposta HTML. O JavaScript do home.js é um forte candidato a isso.

1) Mova a tag <script> que importa o home.js para **fora** do comentário especial de concatenação.

2) Adicione o atributo especial inline à tag, que dispara o plugin:

<script inline src="assets/js/home.js"></script>

3) Rode gulp useref novamente para rebuildar o HTML

Economizamos um request e priorizamos a entrega desse JS: ele já vai embutido direto no HTML e é executado antes de todos os outros.

Como temos um limite de ~6 conexões por hostname, uma técnica para paralelizar os requests é usar mais de um hostname. Para isso, precisamos, claro, de um segundo domínio/subdomínio e precisamos alterar na página.

## Segundo subdomínio

Se você tiver um domínio próprio, pode usá-lo para espelhar os arquivos do nosso site.

Se estiver usando o Google App Engine como sugeri antes no curso, é **muito fácil ter novos subdomínios**. Inclusive, quando fazemos deploy da versão padrão da nossa app, já é possível acessá-la em dois hostnames diferentes:

<http://seu_id_aqui.appspot.com/> <http://web-dot-seu_id_aqui.appspot.com/>

Esse nome web na segunda URL na verdade vem do parâmetro version no app.yaml(que por padrão eu deixei como web).

Um exemplo: meu ID é curso-alura-site e eu mudei o version para assets. Logo posso acessar:

<http://curso-alura-site.appspot.com/> <http://assets-dot-curso-alura-site.appspot.com/>

Você pode usar esse meu <http://assets-dot-curso-alura-site.appspot.com/> se quiser. Ou pode fazer **seu próprio deploy no GAE e usar sua URL** (particulamente interessante se você fez alterações próprias nos arquivos).

## Alterando o HTML

De posse do nosso segundo hostname, basta alterar no HTML os requests que queremos jogar pra esse endereço paralelo. Uma boa ideia é jogar as <img> para carregarem por esse hostname.

Então edite os <img> para ficarem desse tipo:

<img src="http://assets-dot-curso-alura-site.appspot.com/assets/img/busca.svg">

Rode o gulp useref novamente e veja a diferença.

Estudos já mostraram que 2 ou 3 hostnames diferentes é o número ideal. Muitos hostnames paralelos podem causar congestionamento na rede e atrasar o browser. Mas sempre vale testar caso a caso. Às vezes nem compensa ter nenhum hostname a mais.

Para habilitar o cache tbm para o appengine do google no arquivo app.yaml, no final do handlers colocar:

expiration: 365d

Habilitar o cache do servidor apenas para a pasta public e não para o html, o add header serve para avisar o servidor que qualquer usuario pode guardar e acessar o cache, caso seja uma aplicaçao critica deve-se usar o private

location /assets {

expires 1d;

add\_header Cache-Control public;

}

Um cuidado antes de habilitarmos o cache alto no servidor é de garantir que as URLs são únicas e mudarão quando fizermos alterações no site. Fazemos isso adicionando algum tipo de controle de versão no nome do arquivo (um número, ou um timestamp, ou um hash do conteúdo).

Esse processo pode ser manual mas temos o plugin do gulp que já faz isso automaticamente, colocando um hash gerado a partir do conteúdo de cada arquivo.

Nosso gulpfile já foi configurado para isso. Basta rodar: gulp revreplace que todas as tasks anteriores serão rodadas além das novas que fazem as revisões dos arquivos.

Observe o conteúdo gerado e veja como CSS, JS e imagens foram renomeadas.

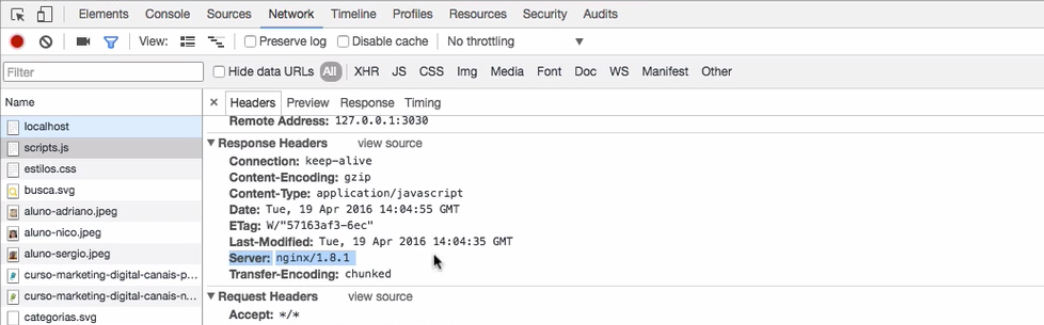
*Importante: renomear os arquivos é um processo destrutivo. Isso quer dizer que não dá pra rodar o revreplace duas vezes. Se precisar rodá-lo novamente, apague o dist, e rode o copy, minify, useref e aí o revreplace de novo.*

No mundo da performance, *request* bom é *request* não feito, por esse motivo fizemos diversas otimizações. Será que não conseguimos ir mais além?

Será que se continuarmos a acessar nossa página, os ícones mudam? E, se houver mudança, será que é frequente?

Estamos tentando chegar no tópico de *cache*. Por padrão, o navegador não faz *cache* e isso ocorre por que podem ocorrer alterações frequentes naquilo que será cacheado, assim, imagine a seguinte situação: o navegador faz *cache*, o site sofre alterações, e o navegador cacheia o conteúdo antigo, assim, quando o usuário acessa ele observará o site antigo.

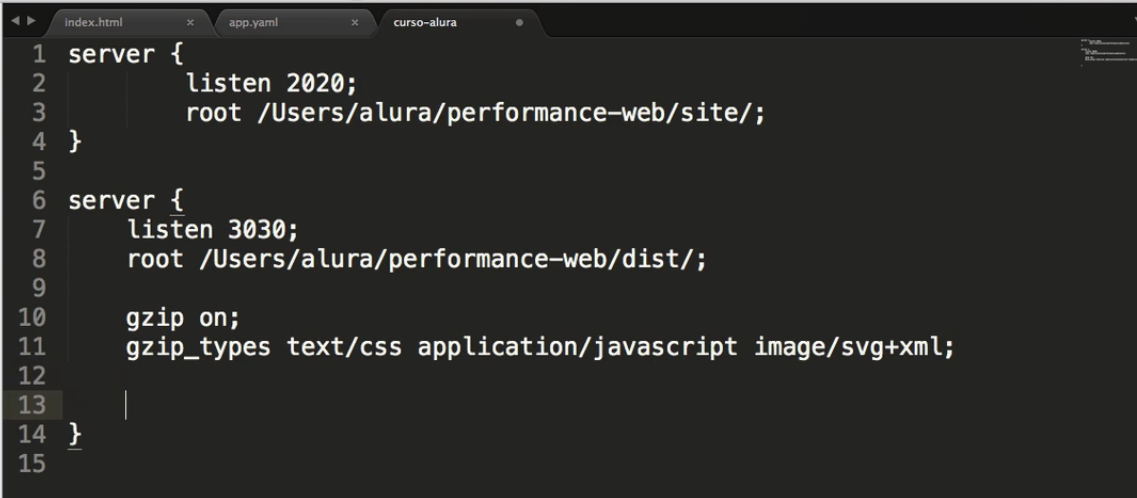
O navegador evita realizar o *cache* a não ser que o servidor instrua o navegador que determinado recurso é "ok" ser cacheado. Esse tipo de configuração fazemos no momento do servidor. O problema é que isso muda de servidor para servidor. A configuração terá que ser feita no momento que o servidor for configurado. Podemos fazer isso através de *headers* de *http*. Quando clicamos na aba *networking*, ele lista os recursos que baixamos e podemos clicar em um deles que ele mostra na aba *headers*todos os cabeçalhos que são enviados tanto no *request* quanto na resposta. Temos aqui diversas informações a respeito da *url*:



A ideia é que podemos adicionar no cabeçalho da resposta outros aspectos, podemos adicionar por quanto tempo o navegador pode cachear o recurso. Esse cabeçalho se chama *expires*.

O que temos que fazer?

Temos que configurar o servidor. Como estamos utilizando o *nginx* vamos configurar diretamente. Vamos em nossa pasta "curso-alura":

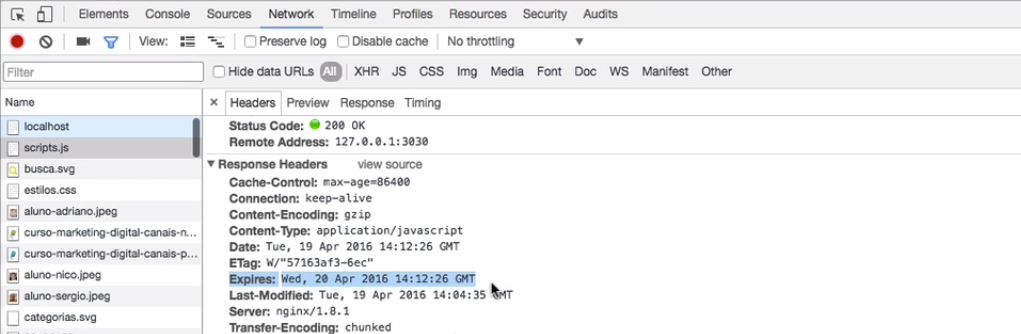


A única coisa que tinhamos habilitado aqui foi o *gzip* e agora vamos habilitar o *expires*. Para isso utilizamos um comando bastante simples, o *expires* e digitamos a quantidade de tempo que queremos que essa habilitação dure. O "1y" refere-se a 1 ano, o "1d" a 1 dia e assim por diante. Digitaremos:

expires 1d

Isso significa que ele vai adicionar, no servidor, o cabeçalho do *response* do cliente. E após termos feito isso vamos recarregar em nosso terminal escrevendo reload -s reload. E vamos acessar, novamente, o navegador.

O que veremos de diferente?

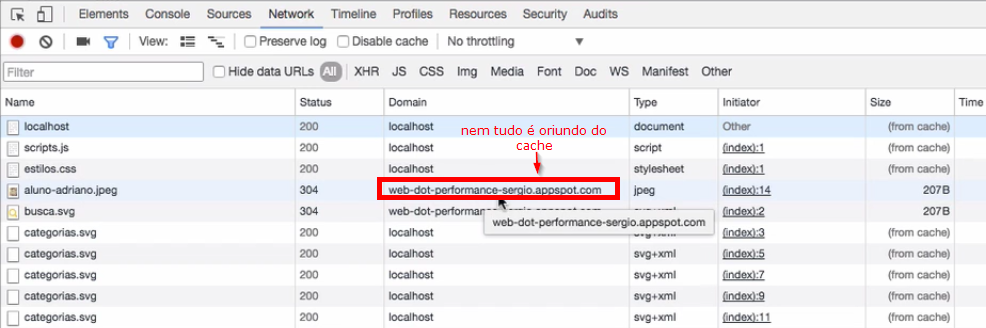


Bom, se observarmos o mesmo arquivo de antes podemos observar algumas diferenças. Por exemplo, temos um *expires*, até quando esse arquivo se manterá vigente, isto é, guardado em *cache*. Podemos reparar que estamos acessando a página em 19 de abril e estamos vendo que o *expires* está configurando para até 20 de abril, tendo duração de um dia, conforme o que tínhamos escrito.

Vamos fazer um teste, vamos atualizar nosso site, mas, atenção, que isso não significa que ele vai baixar as coisas do *cache*. Veremos que ele irá, novamente, no *local host*. Toda vez que o usuário atualiza o navegador, depende na verdade do navegador, ele tenta verificar se não tem atualizações. Para simular uma navegação no *cache* temos que simular uma navegação no usuário. Tipo, clicar em um link. Escondemos um link no logo do Alura, assim, ao clicar nisso faremos uma navegação para a própria *home*.

Fazendo isso e continuando com o *dev tools* aberto podemos ver na coluna *size* que os mesmos arquivos foram cacheados. Nessa coluna teríamos, normalmente, indicado o tamanho da rede utilizada e como ele foi cacheado, não teremos o tamanho da rede. Ele nos diz que usa o *cache* local para baixar o arquivo economizando, assim, diversos *kbytes*.

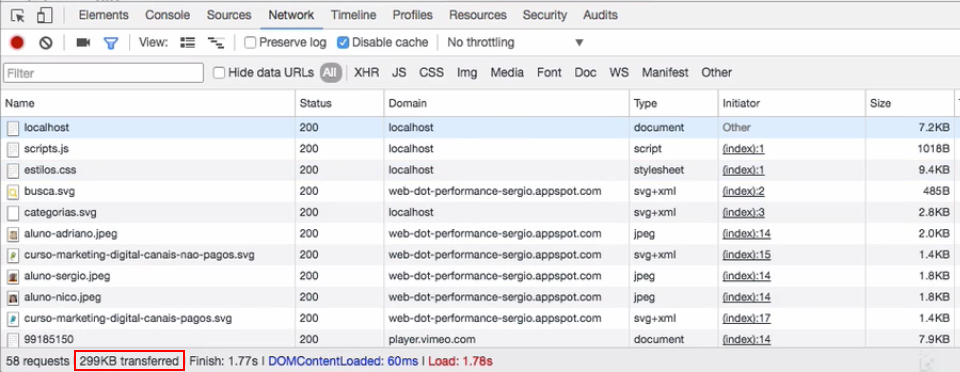
Se repararmos bem, nem tudo tem sua origem no *cache*:



Na verdade nós usamos coisas do domínio paralelo e nele nós não configuramos os *headers*.

Tem alguns detalhes que são importantes: Como testamos se virá sempre do *cache*?

Por exemplo, olharemos lá em baixo do *dev tools* e veremos que 15 *kbytes foram transferidos, mas na verdade, sabemos que é muito mais. Lembra-se da opção*disable*? Se marcarmos essa opção, não interessa o que fizermos agora, sempre que estivermos com o*dev tools*aberto o*cache\* estará disabilitado, então, ele sempre vai baixar os 300 KB repetidamente.

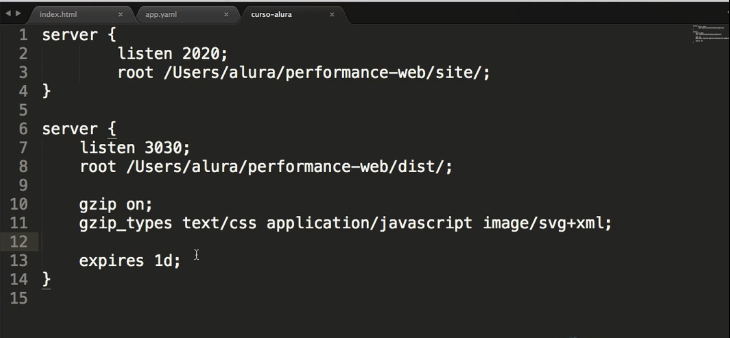


Temos ainda algumas outras coisas importantes que precisamos pensar.

Por exemplo, o *script* é algo que muda com pouca frequência, então, deixar ele com um *cache* de um dia não parece uma opção ruim. O "estilo.css" também não parece ser uma opção ruim. Agora, o *html* seria uma boa?

Ele também foi colocado *header* de *cache*. Mas não é muito boa ideia deixar o *html*cacheado, pois, é algo que alteramos com mais frequência, por exemplo, introduzindo as novidades do dia. Imagine o caso de um *html* de um blog ou algo dinâmico, não são casos que devem ser cacheados.

Vamos reparar que na hora que fizemos a alteração no *nginx* nós digitamos expires 1dque indica que todos os recursos do servidor estejam com *expires 1d*.



Mas, não é bem isso que queremos. Queremos deixar o expires, na verdade, apenas para os arquivos da pasta "assets" que são os arquivos mais estáticos (css, imagens, javascript e etc). No caso do *nginx* é muito fácil fazer isso, é só colocar um bloco location /assets e colocar a configuração do expires dentro disso.

server {

listen 3030;

root /Users/alura/performance-web/dist/;

gzip on;

gzip\_types text/css application/javascript image/svg+xml;

location /assets {

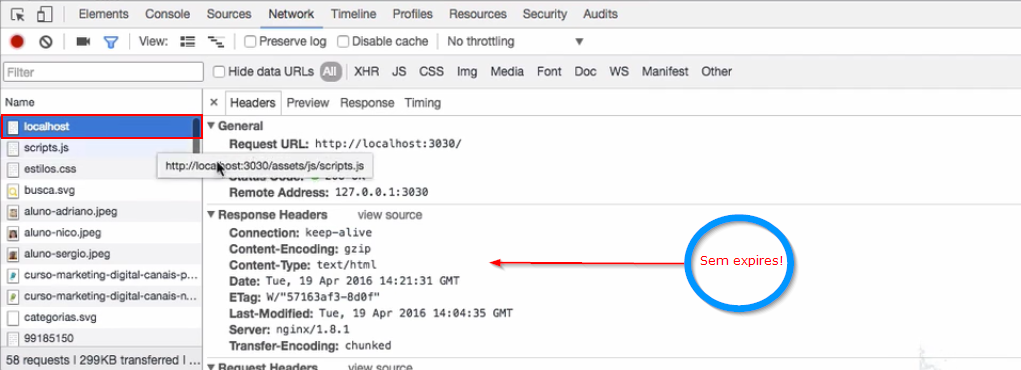
expires 1d;

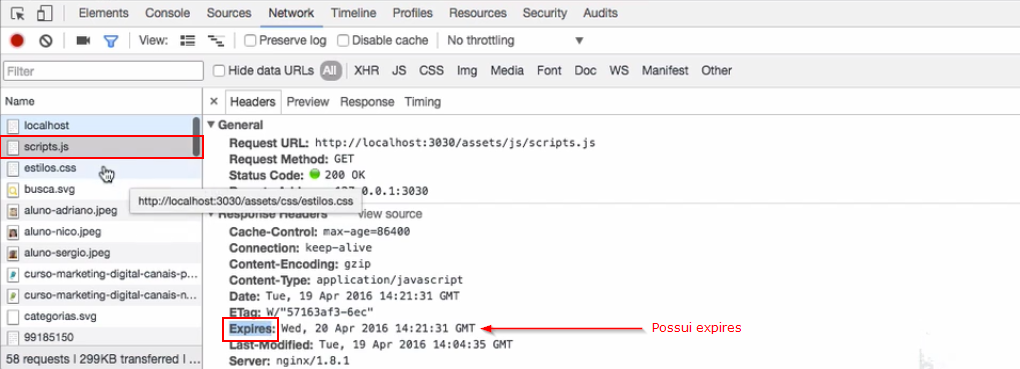
}

}

O que estamos dizendo ao fazer isso é que nesse servidor nem tudo tem expires, apenas o que estiver na localização assets.

Vamos dar um *refrash* no navegador e vamos observar os *headers*. O do *html* não possui mais o expires, mas o "script.js", o "estilo.css" esses, ainda possuem. Conseguimos fazer o que efetivamente queríamos.





Outra coisa que gostaríamos que fosse percebida quando adicionamos o *header expires*, também adicionamos outro *header*, o *cache control*, que possui um atributo *max age*, que significa que ele deve ter um *cache* até daqui um dia. Então, no *cache control*, o tempo deve ser o mesmo que no *expires*. O próprio *nginx* faz isso por nós.

Por que estamos falando disso?

O *cache control* possui mais algumas configurações interessantes. Ele permite além de dizer o quando aquele arquivo expira também dizer quem pode cachear isso por nós. Por padrão, o *cache* foi feito para o navegador pode cachear. Então, acessamos um recurso e o navegador cacheia. Mas a ideia é que entre o navegador e o servidor, existem diversos intermediários na rede.

Talvez, se você trabalha em uma grande empresa, você tenha um *firewall* interno na empresa, ou um *proxy* que pode fazer *cache*. Podemos ter *proxys* em diversos níveis desse tráfego de rede, entre o servidor e o *browser*. A ideia é que se temos um recurso que pode ser cacheado pelo navegador, as vezes esse recurso também pode ser cacheado por outros intermediários, como os *proxys*. O que pode ser uma boa ideia.

Por que estamos falando disso?

Podemos ir ao nosso servidor e adicionar um *header* Cache-Control com valor public:

location /assets {

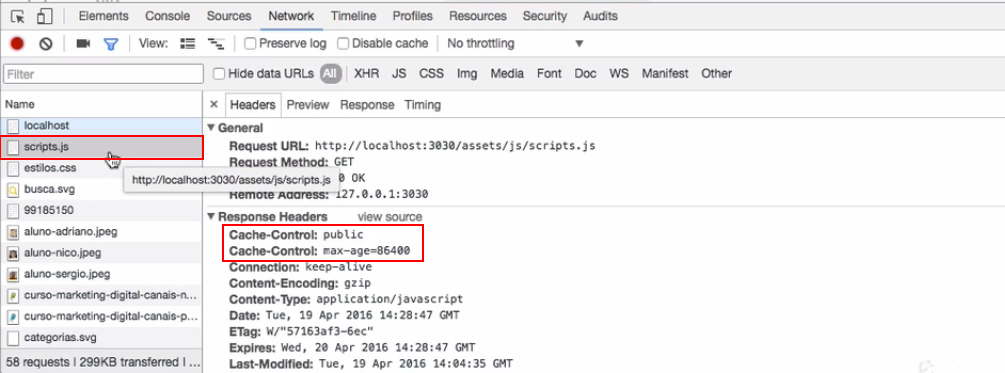
expires 1d;

add\_header public;

}

O que significa um *header* com valor público?

Vamos dar um reload e ir em nosso navegador. Vamos observar o *dev tools*. Podemos reparar que além do nosso recurso possuir um *cache-control* com um período de expiração, temos também um Cache-control que é public.



O publicindica que qualquer intermediário pode cachear esse recurso. O que é muito interessante pois a ideia é que o intermediário mais próximo dele é que vai servir esse arquivo a primeira vez que ele baixar.

Mas em um caso de um recurso que muda de usuário para usuário, então, intermediários não podem cachear. Uma alternativa é mudar para private que significa que apenas o usuário final poderá cachear esse recurso.

Em geral, para *script*, imagens e outros, deixamos *cache-control* no modo public o que indica que qualquer um pode fazer o cacheamento.

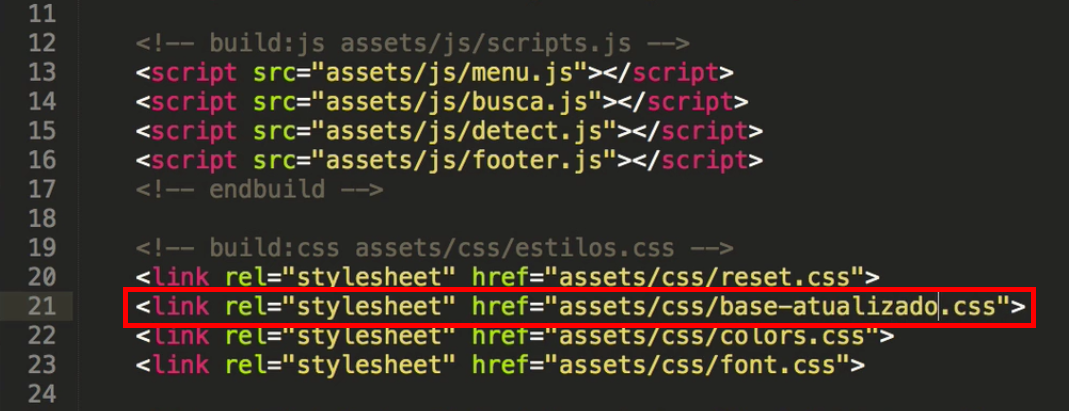
Agora, que já discutimos a questão de como usar o expires, vamos discutir qual seria o tempo ideal para deixar os recursos cachearem. Queremos que nossos recursos sejam o máximo possível cacheados e que um mínimo deles sejam baixados novamente.

Qual seria o problema de deixar o *expires* com um ano de duração? Se mudarmos o site, como o usuário saberá que mudou?

Se o navegador cacheia durante um ano e se mudarmos nossos *css*, *javascript* e etc eles não serão baixados em sua versão atualizada. Na prática temos um conflito, queremos deixar o *expires* alto para evitar que ele faça *download* da mesma coisa várias vezes, mas também queremos que se houver uma modificação, por exemplo, no *css*, que o usuário fique sabendo disso.

Como resolver esse problema?

Quando falamos para um navegador que temos um expires de um ano ele *cacheia* a *url*que foi apontada pelo *html*. Uma técnica que podemos utilizar é a seguinte, deixar o expires alto, como mais ou menos um ano, mas, quando formos fazer alguma alteração, por exemplo, no arquivo "base.css", podemos dar um *rename* e podemos atualizar isso colocando o *rename* do arquivo.

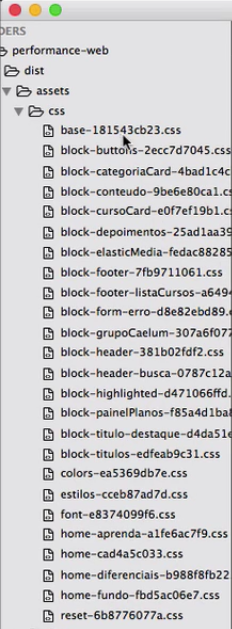


Como estamos juntando tudo no *build* temos que atualizar também o build com estilos-atualizado.css. Então, imagine que quando o usuário acessar o *html*novamente vai baixar o estilos-atualizado, ou seja, ele ignora o que foi cacheado e baixo o novo.

E se quisermos editar um novo arquivo? Teremos que ficar renomeando o build e inventando diversos nome, por exemplo, "mega-atualizado", "super-atualizado" e etc.

A ideia, entretanto, é que não façamos isso na mão. O que queremos fazer é apresentar outra maneira de realizar essa tarefa. Vamos desfazer tudo o que modificamos, pois queremos que tudo seja automatizado.

Até esse momento usamos o *gulp* para realizar automações. Usaremos um *pluggin* dele que pegará todos os *css* da pasta "assets" e irá renomear corretamente quando houver modificação. Esse plugin insere um código ao final dos arquivos e toda vez que houver alteração ele gera outra vez isso. Vamos no Terminal e rodamos o gulp revreplace.



Ele demora um pouco, mas após terminar veremos que todos os arquivos da pasta "css" tiveram seus nomes modificados para ter do lado deles um código. Esse código é de revisão, ele calcula um *hash* para cada arquivo e assim, se o arquivo mudar, alterar-se-a o *hash* também.

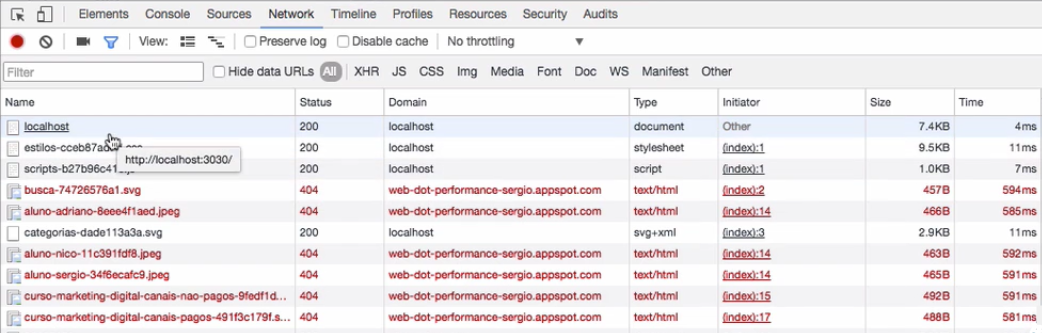
Ele renomeia todos os *css*, imagens e etc.

E como fazemos para renomear o *html* e chamar essa versão renomeada? Ele já faz isso por nós.

Lembre-se que o *html* está minificado então fica difícil visualizar, mas podemos pegar uma *tag* de *script* e veremos que ela está acompanhada do código e isso vale para tudo.

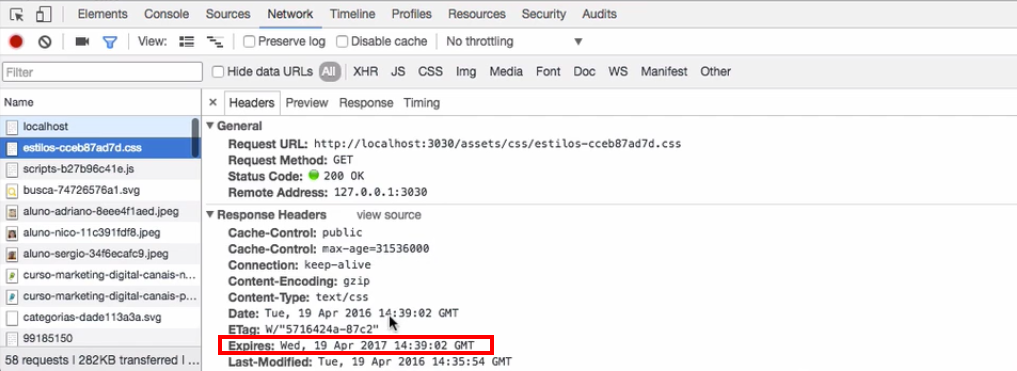
Esse é *pluggin* é muito útil. Se fizermos uma alteração na página, isto é, se mudarmos o conteúdo ele calcula um novo *hash* e coloca isso corretamente no *html*, nos dando mais segurança para utilizar um *expires* de um ano. Assim, podemos deixar o expires com uma validade alta pois sabemos que o pluggin irá gerar uma *url* diferente que será baixada pelo navegador.

Vamos observar no navegador:



Mas repare que temos diversas coisas em vermelho e isso acontece porque ele está tentando baixar conteúdos da nossa *home* secundária. Então, como renomeamos o arquivo, esses arquivos é como se não existissem mais. Temos que fazer o *deploy*. Estamos na nossa "app.yaml" e escrevemos performance-sergio no campo *applications*e quando abrir o aplicativo pedimos para fazer o *deploy*.

Agora, voltou a funcionar:

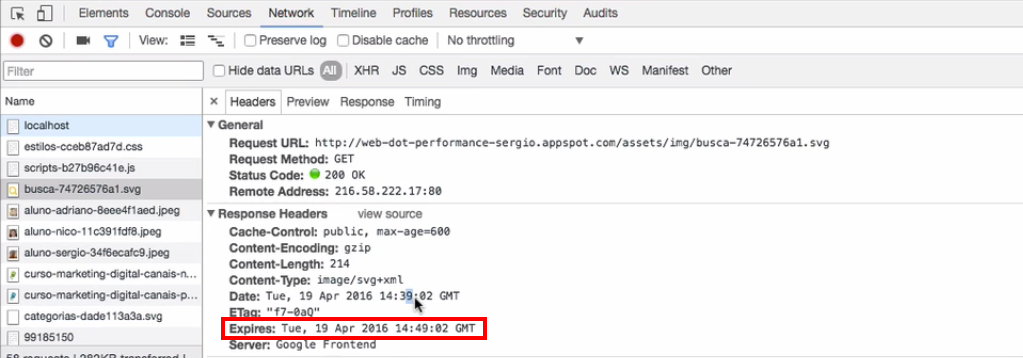


Se analisarmos o navegador veremos que temos o expires até o ano de 2017. Se olharmos o Cache Control teremos a mesma coisa.

Podemos fazer isso com o *html*? Todos os arquivos foram renomeados, com exceção do *index.html*. Faz sentido se tentarmos fazer isso?

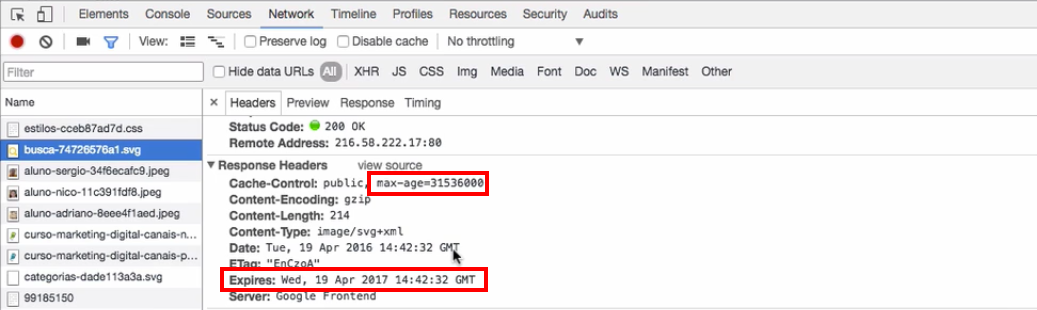
Não, pois o *html* é o ponto de entrada do usuário. A ideia é que a *url* seja estável, ou seja, não deve mudar a cada alteração, por isso, fizemos o expires apenas para os recursos do assets. Deixamos o *html* de lado, isto é, que ele seja expirado daqui um ano, se não, teríamos que gerar novos nomes o tempo todo e o usuário ficaria sem saber o que digitar.

Um último ponto que fizemos foi a configuração do *header* no *nginx* local. Já vimos que nem todos os recursos são baixados do *nginx* local, alguns são baixados de nosso link secundário. Lembrando que estamos utilizando o *app engine* e por padrão ele coloca um *expires*, que é bem baixo, na verdade,



E se quisermos alterar o *expires* no *app engines*?

Lembra-se do "app.yaml"? Temos nele a pasta "assets". Em baixo dela colocaremos um atributo expiration e a quantidade de tempo que desejamos que seja o tempo até expirar. Importante notar que a configuração funciona com dias. Após isso, fazemos o *deploy* e podemos conferir se funcionou:



Acabamos de mostrar dois servidores diferentes, o *google app engine* e o *nginx*. Como fazemos a configuração de expiração em *apache*, *is* e etc ? Geralmente essa configuração é um comando curto.

Recapitulando, o importante é colocar uma "expiração" longa para os recursos estáticos da página. É preciso tomarmos cuidado para que possamos atualizar isso depois e uma maneira de fazer isso é agregando um número de versão para esse recurso. Por exemplo, o *gulp* é capaz de fazer isso, mas se você quiser utilizar outra maneira pode pesquisar como fazer automação usando uma *url* diferente baseada no conteúdo dela.

O é importante lembrar de se for inserir um *expires* alto, é preciso inserir algo para alterar a *url* se houver alteração de conteúdo.