

Performance Web II: Critical Path, HTTP/2 e Resource Hints: Aula 7 - Atividade 3

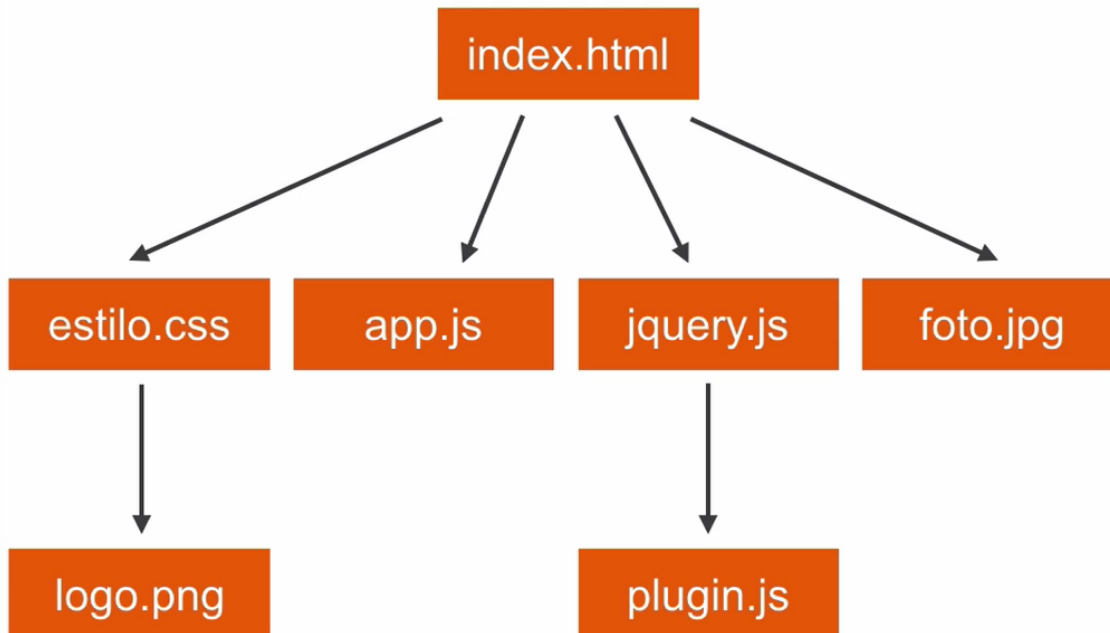
HTTP/2 avançado | Alura

9-12 minutes

Tínhamos mencionando que o *inline* de recursos é importante para priorizar conteúdos. Mas isso dá trabalho, com isso perdemos o *cache*, necessitamos de um *plugin* do gulp. Idealmente, gostaríamos que tudo fosse separado em arquivos distintos para ficar mais fácil de trabalhar. Aqui temos um caso de exemplo:

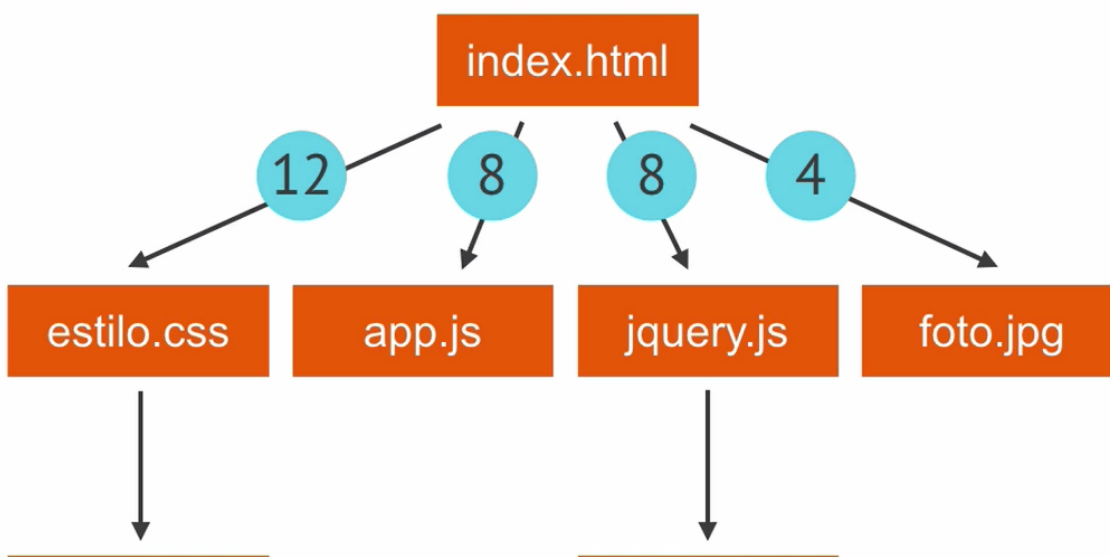
```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="estilo.css">
  <script src="app.js" async></script>
</head>
<body>
  
  ...
  <script src="jquery.js"></script>
  <script src="plugin.js"></script>
</body>
</html>
```

E se nos perguntarmos, além do HTML, nesse caso, qual seria a segunda coisa mais essencial de ser baixada? É o "CSS" pois ele bloqueia a renderização, o "jQuery" é bastante importante, mas como ele está no final da página, ele perde um pouco de sua importância. A questão é, o navegador é sábio, ele consegue distinguir a importância de cada coisa e a relação entre elas. Ou seja, o navegador compreende que existe uma relação de prioridade:



priorizar conteúdo

Ele busca fazer a renderização na ordem certa, mas claro, alguns azares podem ocorrer, por exemplo, usar uma conexão que está mais lenta por algum motivo para baixar o "CSS", por exemplo faz com que o *jQuery* possa chegar antes, o que acaba por não fazer muito sentido. O navegador tem pouco controle sobre isso no HTTP 1, mas no HTTP/2 isso já muda. Ele consegue fazer exatamente o que temos na imagem a cima e, ainda, é interno desse protocolo comunicar as dependências que existem entre as coisas. Se o *plugin*, por exemplo, depende do "jQuery" ele não será carregado antes. Além disso, ele consegue comunicar um peso diferente para cada uma das requisições e dessa forma o navegador irá poder decidir o que fazer:

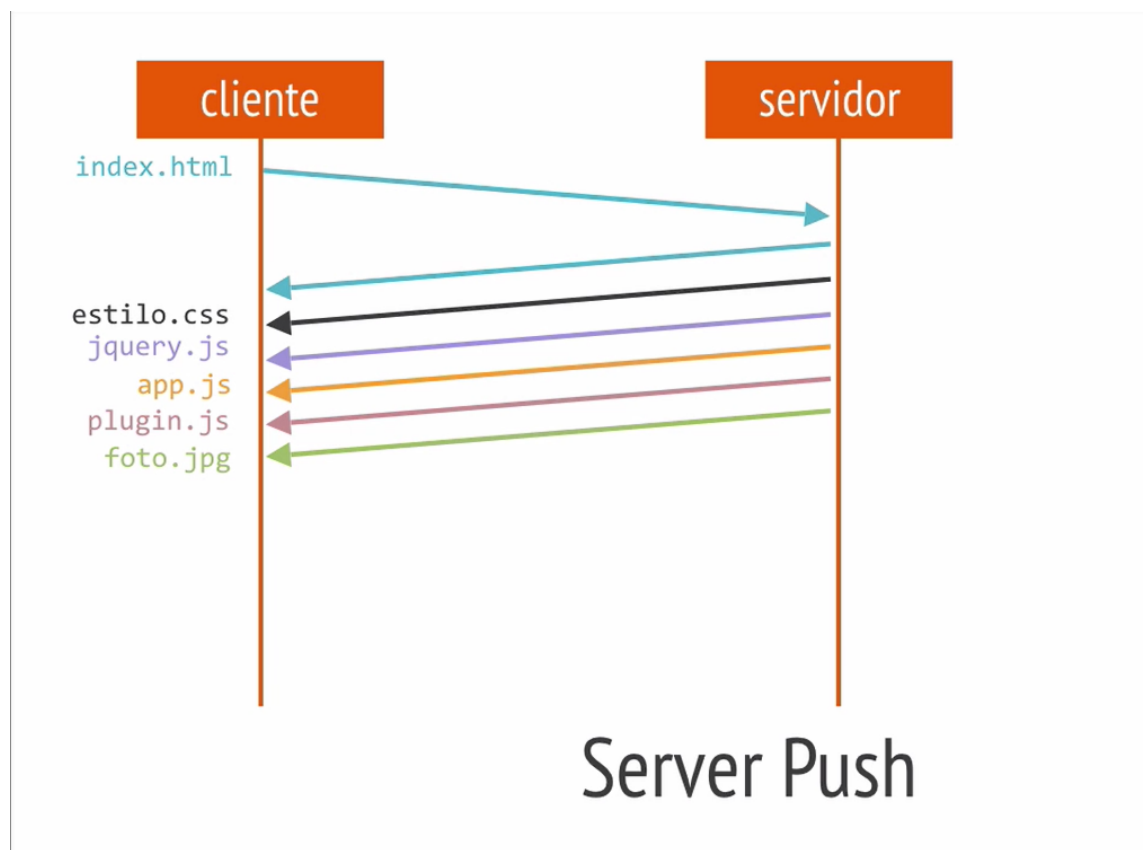


logo.png

plugin.js

priorizar conteúdo

Assim, o que se faz é comunicar tanto a dependência quanto a prioridade de cada uma das requisições e o servidor utilizará todas essas informações para carregar na melhor ordem possível. De antemão, com o conhecimento da importância de algo, por exemplo, do "CSS", ao pedirmos o "HTML" ele já manda o "CSS" junto. Isso é denominado *Server Push* no HTTP/2. Quando o cliente pede o "HTML" o servidor já sabe da importância do servidor e envia o "CSS" de imediato. Observe:



Isso significa que diversas respostas do servidor são enviadas antes da requisição. O cliente ainda nem fez a requisição, depois é que ele vai realizar uma espécie de requisição fantasma. O servidor empurra coisas que ele sabe que são necessárias, pela primeira vez um protocolo dá respostas empurradas sem necessariamente serem requisitadas, esse é exatamente o que o *inline* faz, só que com o *server push* podemos fazer isso de maneira transparente no protocolo, sem precisar enfiar o "CSS" no "HTML". O "CSS" continuará a ser uma

resposta independente, poderá ser cacheado, ele controla isso e se quiser pode ser cancelado.

O *Server Push* é cancelável, o cliente pode pedir para que isso pare de ser mandado. Tudo isso de maneira assíncrona, ele não espera confirmação, ele começa a mandar e o navegador pode falar "pare, que já temos". A ideia é que é possível cancelar isso e usar o *cache*. São os mesmos benefícios do *inline*, mas sem seus aspectos ruins.

O *Server Push* possibilita:

- priorização
- cancelável
- cacheável

Estamos vendo como funciona o protocolo por cima, veremos como ele funciona exatamente usando uma implementação de HTTP/2.

Com isso, podemos compreender as principais contribuições do HTTP/2 para o mundo da web, do front-end. Temos um protocolo inteiro que é:

- binário
- criptografado
- podemos usar compressão
- multiplexação das requisições e respostas
- priorização, é possível comunicar ordens e dependências ao servidor
- uso do *Server Push*

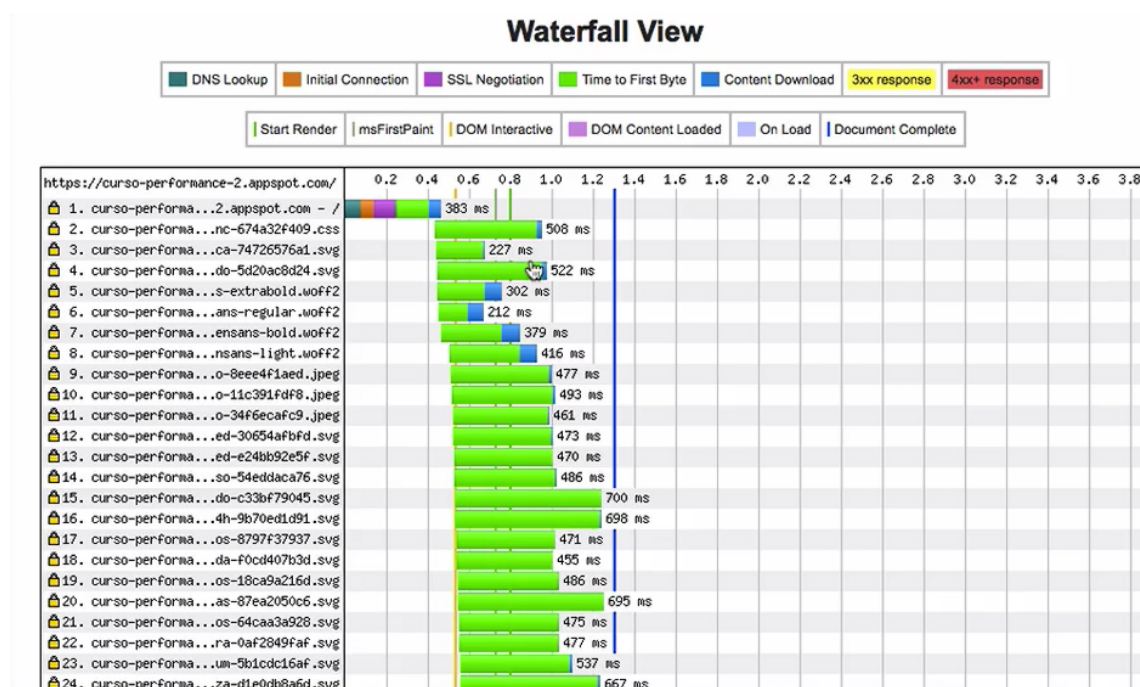
E isso que faz com que nossa performance melhore, principalmente, quando utilizamos redes *mobile*, no "3G" onde temos uma rede com alta latência e onde o protocolo é otimizado para esse cenário. Ele é um ambiente mais fácil de desenvolver e ele é compatível hoje com todos os servidores e navegadores modernos. Se você estiver usando um navegador antigo ele irá fazer o *fall back* para HTTP 1, assim, ele não quebra a internet para fazer isso. Isso deixa o protocolo mais leve, por exemplo, o servidor fica mais leve, não é mais necessário seis

conexões, basta uma. E, também, ele é mais seguro por que possui o https obrigatório na implementação atual.

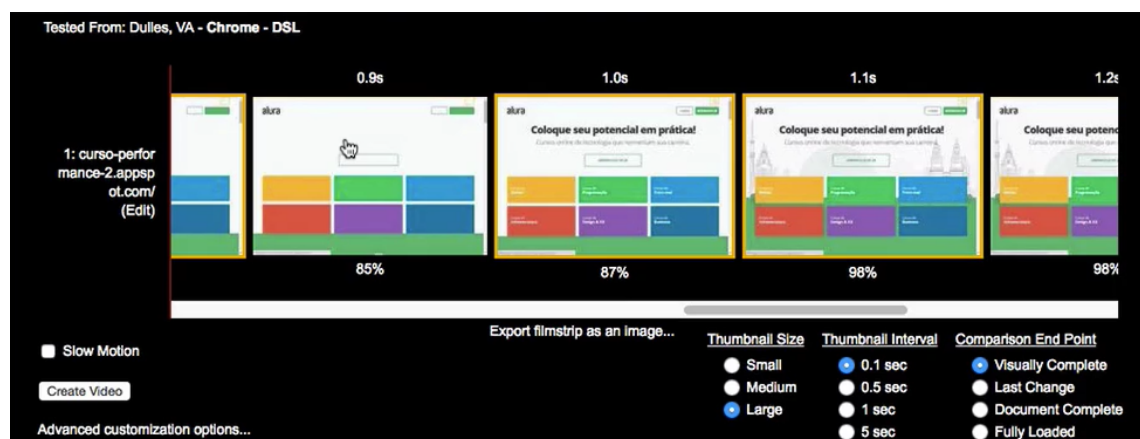
Vários site, por exemplo, o **Facebook** utilizam o HTTP/2.

Server Push na prática

Agora que discutimos sobre a teoria do HTTP/2 vamos testar seu uso na prática. Rodamos no *Web Page Test* a última versão do site e conseguimos ver no gráfico de *Water Fall*:



Esse gráfico traz uma série de informações, Todas as barras da cor "verde limão" indicam o tempo de espera. Após baixar o "HTML" é que são iniciados os demais downloads. Enquanto o "HTML" não é baixado não é possível iniciar os outros recursos. No *Web Page Test* podemos simular um vídeo da renderização da página:



Nessa execução podemos perceber que o site está bastante rápido.

Mas, se observarmos veremos que o fundo, o arquivo "background-cidade-fundo" tarda um pouco a ser carregado, se voltarmos ao gráfico *Water Fall* podemos verificar isso também. Vamos testar com o fundo, na prática, o **Server Push**.

Para fazer isso temos que abrir nosso código. A implementação do **Server Push** é distinta em cada servidor, ela não faz parte do protocolo. Precisamos aprender a fazer isso no servidor que estamos utilizando. Para facilitar estávamos utilizando o *Google App Engine*. Então, a nossa pergunta, na verdade é, como fazer para suportar o **Server Push** no **Google App Engine**? A maior parte dos servidores modernos acabaram acordando que a melhor maneira de trabalhar com **Server Push** é na hora em que dermos a resposta ao "HTML", incluir um *header* na resposta que instrua o servidor a enviar algo com **Server Push** logo em seguida.

Para fazer esse *header*:

```
Link: <assets/img/background-cidade-fundo.svg>;  
rel=preload; as=image
```

Na verdade, esse *header* é muito parecido ao link "preload" que tínhamos feito anteriormente para carregar o CSS assíncrono. Esse cabeçalho HTTP que criamos vai ser lido pelo servidor e o servidor vai transformar isso em um *Server Push* no HTTP/2.

Para passar isso para o **Google App Engine** não podemos usar a *tag* HTML. Temos que fazer isso antes, no *back-end*. Se você trabalha com PHP basta abrir a tag "php", abrir um *header* e colocar o conteúdo nele:

```
<?php  
header("Link: <assets/img/background-cidade-fundo.svg>;  
rel=preload; as=image");  
?>
```

Se você trabalha com *Java* usamos o *response*, o *addHeader* e o valor. Exemplo:

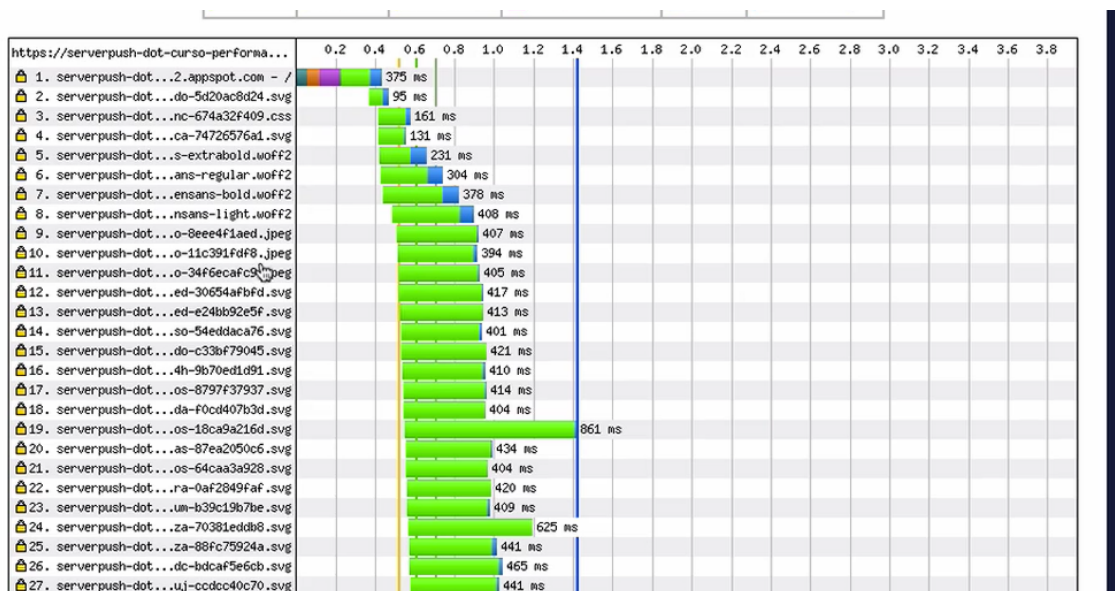
```
response.addHeader(...)
```

Na verdade, depende da linguagem com a qual você estiver trabalhando no *back-end*. Se você estiver trabalhando com o "HTML" puro. Vamos mostrar como se faz isso no **App Engine**, mas na verdade em qualquer outro *back-end* seria da mesma maneira, só temos que cuidar a linguagem.

Abrimos o arquivo "App.yaml" e acrescentamos uma configuração a mais, no *handlers*, a `http_headers` que permite, justamente, adicionar isso na resposta:

```
1 # Arquivo de configuração para deploy no Google App Engine que veremos na aula
2
3 runtime: php55
4 api_version: 1
5 version: web
6 application: curso-performance-2
7
8 handlers:
9   - url: /
10     static_files: index.html
11     upload: index.html
12     expiration: 0s
13     http_headers:
14       Link: <assets/img/background-cidade-fundo.svg>; rel=preload; as=image
15
```

Vamos "rebuildar" o projeto e, agora, testar. Vamos fazer o "deploy" dele. Vamos rodar, novamente, no *Web Page test* e observaremos o *Waterfall*:

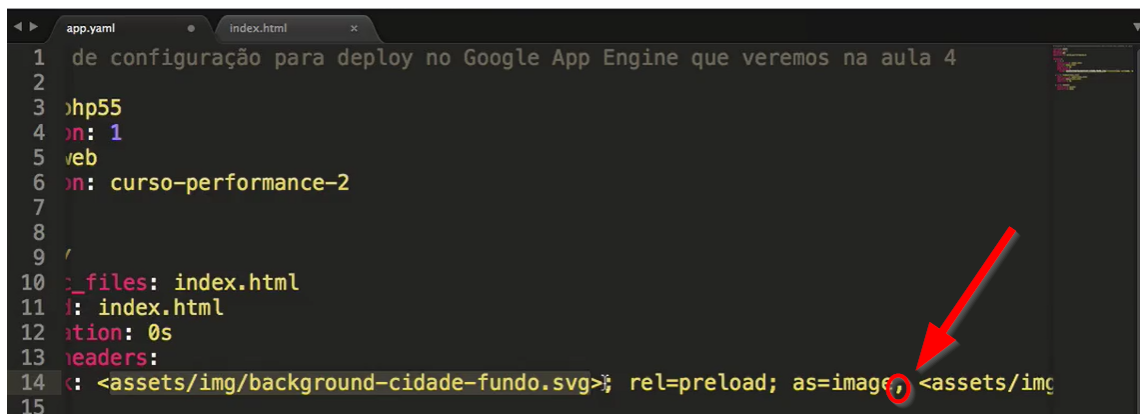


Repare a diferença, o primeiro *request* depois do HTML é justamente o "background-cidade-fundo". Observe a diferença entre o tempo de baixar o "HTML" (cor azul) e começar a baixar o "background" do fundo. Eles estão praticamente grudados. Todos os outros *requests*

que não estão com *server push* não estão com o azul emendado. Se observarmos o *Film Strip* veremos que ocorre uma mudança de tempo, agora, ele está bem mais rápido:

Visualmente podemos constatar que houve uma melhoria utilizando o *Server Push*. Quando mencionamos o uso do *Server Push* mencionamos que ele existia para resolver o problema do *inline*.

Por exemplo, o logo do alura que está "inlineado", possui a desvantagem de perder o *cache* e do código ficar muito confuso. Você pode fazer o **Server Push** do logo também, basta retirar o logo do "inline" e copiar o seu "assets/img/logo-alura.svg" que está no "index.html" e colocar ele no "app.yaml" separando-o do "background-cidade-funco" usando uma vírgula (,).



```
1 de configuração para deploy no Google App Engine que veremos na aula 4
2
3 runtime: php55
4 version: 1
5 service: web
6 runtime: curso-performance-2
7
8
9
10 _files: index.html
11 _files: index.html
12 timeout: 0s
13 readers:
14 - <assets/img/background-cidade-fundo.svg>; rel=preload; as=image, <assets/img
```

A ordem também é importante, então, como queremos que o logo apareça antes do fundo apenas colocamos ele na frente do "fundo." Fazendo o *deploy* podemos abrir uma nova página, escrevendo "https" para indicar que é HTTP/2. E, agora, abrimos o *dev tools*, marcamos a opção *disable cache* e vamos escolher um *throttling 3G* e vamos atualizar a página e, agora, observando o *console* podemos verificar dois *requests* meio esquisitos. Eles são diferentes porque são iniciados por **Server Push**, ou seja, estão disponíveis para serem renderizados muito antes dos demais recursos.

A ideia é que através de um cabeçalho simples, podemos instruir o servidor a empurrar essas coisas para o cliente o mais rápido possível antes de disparar qualquer *request*. A vantagem é fazer o **server push** ao em vez de *inline* e isso fica a seu critério.

