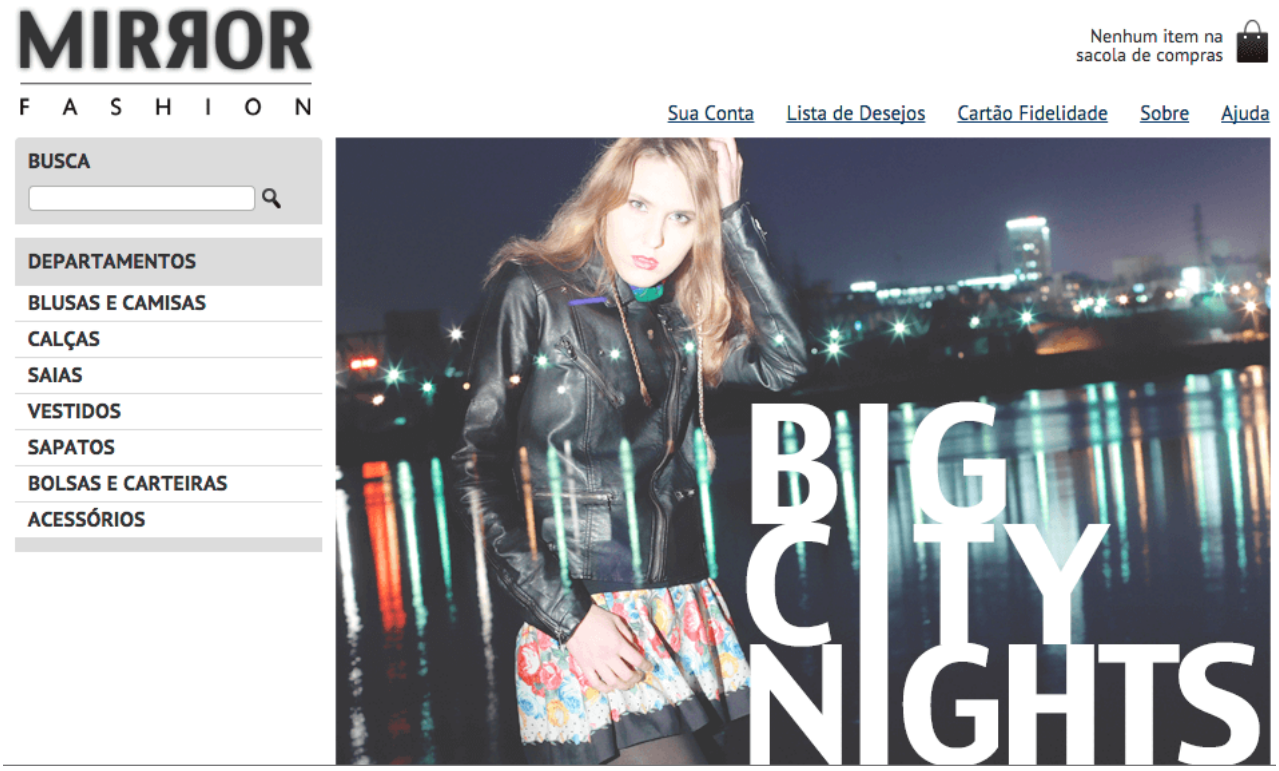
06

Lidando com latência e banda

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/gulp/stages/03-capitulo.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo. Dentro da pasta **projeto**, não esqueça de executar no terminal o comando **npm install** para baixar novamente todas as dependências.

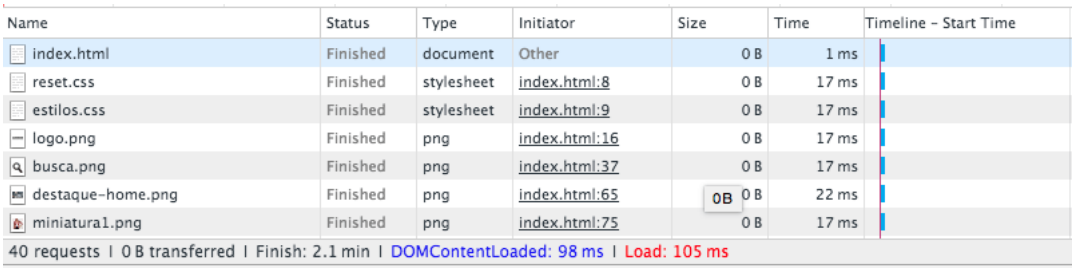
Muito bem, já temos o processo de otimização de imagens pronto, inclusive não realizamos mais um processo destrutivo, pelo contrário, mantemos as imagens originais para que possam ser retocadas quando necessário. No final das contas, o que fizemos foi dar uma melhor experiência para o usuário, onerando menos sua largura de banda fazendo com que nossas imagem sejam carregadas mais rápido. E agora, qual o próximo passo?

Vamos abrir a página `projeto/src/index.html` em seu navegador, de prefência **Google Chrome**, pois é este que está sendo utilizado. Essa é a página principal:



Mesmo com a imagem otimizada, pode ser que algum usuário ainda ache a página lenta, mesmo que sua franquia de internet tenha uma largura de banda generosa. Isso significa, que não é apenas o tamanho dos arquivos que influenciam no tempo de carregamento de uma página. Então, o que mais influencia?

Com a página sendo visualizada, abra o console do seu navegador. Há uma aba chamada **rede** ou **network**, dependendo da língua utilizada pelo seu navegador. Clicando nesta aba, recarregue a página. Você verá uma estatística de todos os arquivos que foram carregados:



Name	Status	Type	Initiator	Size	Time	Timeline – Start Time
index.html	Finished	document	Other	0 B	1 ms	
reset.css	Finished	stylesheet	index.html:8	0 B	17 ms	
estilos.css	Finished	stylesheet	index.html:9	0 B	17 ms	
logo.png	Finished	png	index.html:16	0 B	17 ms	
busca.png	Finished	png	index.html:37	0 B	17 ms	
destaque-home.png	Finished	png	index.html:65	0 B	22 ms	
miniatura1.png	Finished	png	index.html:75	0 B	17 ms	

40 requests | 0 B transferred | Finish: 2.1 min | DOMContentLoaded: 98 ms | Load: 105 ms

Veja, que mesmo abrindo a página localmente, além de termos carregado o `index.html` o navegador precisa baixar também todos os recursos externos, aqueles que não fazem parte do corpo do HTML como os arquivos `.css`, `.js`, inclusive as imagens. Nossa página realiza um total de 24 requisições!

index.html

```
<link rel="stylesheet" href="css/reset.css">
<link rel="stylesheet" href="css/estilos.css">
<link rel="stylesheet" href="css/mobile.css" media="(max-width: 939px)">
<img><! -- várias imagens -->
<script src="js/jquery.js"></script>
<script src="js/home.js"></script>
```

O problema das múltiplas requisições

Mas qual o problema de realizarmos várias requisições? Cada navegador possui um número máximo de requisições simultâneas para um mesmo domínio, aliás, bem reduzido em smartphones. Se chegarmos a este limite, o próximo recurso a ser baixado terá que esperar até que uma das requisições acabe. Isso dá a sensação de carregamento lento da página para o usuário.

Outro ponto é a latência, que geralmente afeta bastante redes de dispositivos móveis. A latência é o tempo entre você realizar uma requisição ao servidor e ela começar a ser respondida. Redes 3G sofrem bastante de latência.

Não podemos contar que nossos usuários tenham a rede com a menor latência do mundo ou um navegador que faça zilhões de requisições simultâneas para um mesmo domínio. É por isso que precisamos otimizar nossas páginas, diminuindo sempre que possível o número de requisições.

A técnica merge (concatenação)

Para reduzir o número de requisições feitas pelo navegador, podemos juntar arquivos de um mesmo tipo num só arquivo. Na nossa página, por exemplo, poderíamos juntar todos os arquivos `.css` em um único arquivo, a mesma coisa para todos os arquivos `.js`. Na hora de o navegador interpretar o CSS, não importa se são dois arquivos ou um só. É claro que não podemos fazer isso com nossos arquivos originais, a separação ajuda na manutenção do nosso código. Essa técnica de juntar arquivos de tipos semelhantes em um único arquivo é chamada de **merge** ou **concatenação**. Vamos aplicá-la ao nosso projeto.

Vamos juntar todos os arquivos CSS em um arquivo chamado `all.css` e todos os arquivos JS utilizados em um arquivo chamado `all.js`. Não podemos nos esquecer de alterar `index.html` e apontar para os novos arquivos:

```
<!-- projeto/dist/index.html -->
<!DOCTYPE html>
<html>
```

```
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>Mirror Fashion</title>
    <link rel="stylesheet" href="css/all.css">
  </head>
  <body>
    <!-- código posterior omitido -->
    <script src="js/all.js"></script>
  </body>
</html>
```

Com esse processo, poupamos três requisições, mas imagine um projeto maior com vários arquivos CSS e JS? Com certeza melhoraríamos a velocidade de carregamento da página. Pode parecer pouco, mas caso nossa página tivesse 10 css's evitaríamos de realizar 9 requisições, o que com certeza aceleraria bastante o carregamento da página.

Mas já pensaram em ter que repetir esse processo toda vez que um arquivo for alterado ou um novo arquivo for introduzido? Teríamos um trabalhão! Só alteramos o `index.html`, mas todas as outras páginas devem apontar para `all.css` e `all.js`. Mais uma vez o Gulp nos ajudará neste processo.

Automatizando a concatenação/merge

Vamos realizar a concatenação automática de scripts através do plugin gulp-concat (https://github.com/contra/gulp-concat). Instalando-o no terminal com o npm:

```
npm install gulp-concat@2.6.0 --save-dev
```

Vamos importar o plugin em nosso `gulpfile.js` e criar a tarefa `build-js`, mas ainda sem qualquer ação:

```
// importou gulp-concat
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin')
  ,clean = require('gulp-clean')
  ,concat = require('gulp-concat');


gulp.task('copy', ['clean'], function() {
    return gulp.src('src/**/*')
        .pipe(gulp.dest('dist'));
});

gulp.task('clean', function() {
    return gulp.src('dist')
        .pipe(clean());
});

gulp.task('build-img', ['copy'], function() {

  gulp.src('dist/img/**/*')
    .pipe(imagemin())
    .pipe(gulp.dest('dist/img'));
});
```

```
gulp.task('build-js', function() {
    // nossa tarefa que nada faz ainda
});
```

Qual será nosso fluxo de leitura? Todos os arquivos dentro de `dist/js` . E o nosso destino? A pasta `dist/js` :

```
// código anterior omitido

gulp.task('build-js', function() {
    gulp.src('dist/js/**/*.js')
        .pipe(gulp.dest('dist/js'));
});
```

Nosso fluxo de leitura lê diversos arquivos, mas é a tarefa concat que realizará a concatenação de todos eles. É o resultado de concat que será passado para o fluxo de destino, gravando assim um arquivo apenas. Vamos adicionar a concatenação entre os dois fluxos. A tarefa recebe como parâmetro o nome do arquivo resultante da concatenação:

```
// código anterior omitido
gulp.task('build-js', function() {
    gulp.src('dist/js/**/*.js')
        .pipe(concat('all.js'))
        .pipe(gulp.dest('dist/js'));
});
```

Será que funciona? Primeiro, vamos rodar a tarefa `copy` e depois `build-js` :

```
npm run gulp copy
npm run gulp build-js
```

Excelente, o arquivo `dist/js/all.js` foi criado, porém ainda temos um problema. Se abrirmos a página `dist/index.html` vemos que ele ainda continua apontado para os arquivos JS separados. Precisamos automatizar também esse processo de alteração através do plugin gulp-html-replace (https://github.com/VFK/gulp-html-replace).

Baixando:

```
npm install gulp-html-replace@1.5.4 --save-dev
```

E como de costume, não podemos nos esquecer de fazer o `require` do módulo dentro do nosso `gulpfile.js` . Vamos aproveitar e criar uma tarefa chamada `build-html` , mas ainda sem ação alguma:

```
// importou gulp-html-replace
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin')
  ,clean = require('gulp-clean')
  ,concat = require('gulp-concat')
  ,htmlReplace = require('gulp-html-replace');


gulp.task('copy', ['clean'], function() {
```

```
    return gulp.src('src/**/*')
        .pipe(gulp.dest('dist'));
});

gulp.task('clean', function() {
    return gulp.src('dist')
        .pipe(clean());
});

gulp.task('build-img', ['copy'], function() {

  gulp.src('dist/img/**/*')
    .pipe(imagemin())
    .pipe(gulp.dest('dist/img'));
});

gulp.task('build-js', function() {
    gulp.src('dist/js/**/*.js')
        .pipe(concat('all.js'))
        .pipe(gulp.dest('dist/js'));
});

gulp.task('build-html', function() {
    // sem ação alguma, por enquanto
});
```

Nosso fluxo de leitura será todos os nosso arquivos HTML e nosso destino será o diretório com esses arquivos:

```
// código anterior omitido

gulp.task('build-html', function() {

    gulp.src('dist/**/*.html')
    .pipe(gulp.dest('dist/'));
});
```

Sabemos que o módulo `htmlReplace` deve vir entre o fluxo de leitura e o fluxo de escrita. Mas como ele saberá quais arquivos JS juntar e qual o nome do arquivo final? Precisamos colocar uma meta informação em todas as páginas que desejamos que o htmlReplace opere. Essa meta informação é através de um comentário especial. Vamos adicioná-lo por enquanto apenas em `index.html` :

```
<!-- build:js -->
 <script src="js/jquery.js"></script>
 <script src="js/home.js"></script>
 <!-- endbuild -->
```

Veja que este comentário especial tem uma estrutura que delimita seu bloco, começando com `<!-- build:js -->` e terminando com `<!-- endbuild -->` . Tudo que estiver neste bloco é processado pelo `htmlReplace` . Queremos que ele troque todo o conteúdo de `build:js` pelo arquivo `all.js` que criamos em nossa tarefa que trabalha com scripts:

```
// código anterior omitido
gulp.task('build-html', function() {
```

```
    gulp.src('dist/**/*.html')
        .pipe(htmlReplace({
            'js': 'js/all.js'
        }))
        .pipe(gulp.dest('dist/'));
});
```

Perceba que o `htmlReplace` recebe um objeto como parâmetro com a chave `js`. Essa chave equivale ao comentário `build:js`. Se a chave fosse `almeida`, lá no comentário usaríamos `build:almeida`. Veja que seu valor é o nome do arquivo que substituirá o bloco do comentário no HTML.

Vamos testar o resultado:

```
npm run gulp copy
npm run gulp build-img
npm run gulp build-html
npm run gulp build-js
```

Agora, cuidado para não nos confundirmos, vamos abrir o arquivo **dist/index.html**. Observando o arquivo, vemos que os dois scripts foram trocados por um:

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

O problema é que ainda estamos executando cada tarefa em separado no terminal. Aliás, precisamos repensar a execução de nossas tarefas, inclusive verificar quais podem correr em paralelo. A tarefa `clean` precisa executar e nenhuma outra pode executar enquanto ela não terminar de apagar a pasta `dist`. A tarefa `copy` também, nenhuma outra tarefa pode ser executada enquanto ela não terminar. A partir de agora, tanto `build-html` quanto `build-js` podem ser processadas em paralelo. Não faz mal se `build-html` terminar depois de `build-js` ou vice-versa. Então, como resolver?

Vamos criar uma tarefa padrão (default) que será executada pelo Gulp se nenhum tarefa for passada como parâmetro. Vamos aproveitar e adicioná-la como primeira tarefa do nosso `gulpfile.js`:

```
// importou gulp-concat
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin')
  ,clean = require('gulp-clean')
  ,concat = require('gulp-concat')
  ,htmlReplace = require('gulp-html-replace');

gulp.task('default', ['copy'], function() {

});

// código posterior omitido
```

Antes de continuarmos, vamos remover a dependência de `copy` da nossa tarefa de otimização de imagens, que também pode ser executada em paralelo depois da pasta `dist` tiver sido copiada:

```
// código anterior omitido
// a tarefa não depende mais de copy
gulp.task('build-img', function() {

  gulp.src('dist/img/**/*')
    .pipe(imagemin())
    .pipe(gulp.dest('dist/img'));
});
// código posterior omitido
```

A grande sacada é fazermos com que a tarefa padrão processe as tarefas `clean` e `copy` sincronamente, o que já está fazendo, mas que também execute as tarefas `build-img` , `build-html` e `build-js` em paralelo. Para isso usamos o método `gulp.start` e nele passamos todas as tarefas que queremos executar:

```
// código anterior omitido
gulp.task('default', ['copy'], function() {
    gulp.start('build-img', 'build-html', 'build-js');
});
// código posterior omitido
```

Agora, só rodar no terminal:

```
npm run gulp
```

Perfeito! Funcionou: o arquivo foi criado e o HTML modificado. Que tal abrirmos `dist/index.html` e verificarmos se tudo funciona? Hum, os botões `mostrar mais` sumiram. Isso aconteceu porque houve um erro no JavaScript: `$ is not defined` . Este erro acontece quando tentamos usar o script do jQuery antes dele ter sido carregado. O problema é que o processo de concatenação deveria ter adicionado primeiro o jQuery antes de concatenar com os demais scripts que o utilizam. E agora?

Podemos passar como parâmetro para o concat uma lista de com todos os arquivos que ele precisa concatenar e dessa maneira podemos colocar o `jquery.js` como primeiro. Alterando nossa tarefa:

```
// código anterior omitido

gulp.task('build-js', function() {
    gulp.src(['dist/js/jquery.js',
      'dist/js/home.js',
      'dist/js/ativa-filtro.js'])
        .pipe(concat('all.js'))
        .pipe(gulp.dest('dist/js'));
});
```

Rodando novamente:

```
npm run gulp
```

Agora sim, nosso projeto da pasta `dist` funciona! Antes de fazermos a mesma coisa com nossos arquivos CSS, há outra otimização que ainda podemos fazer.

O problema de grandes arquivos

Vimos que não apenas nossas páginas, mas todos recursos externos como imagens, arquivos CSS e JS também são baixados. Trabalhamos o problema da latência, mas podemos melhorar ainda mais a velocidade de carregamento de nossas páginas otimizando a quantidade de dados enviados.

Quando contratamos uma franquia de internet temos acesso a uma largura de banda que determina a velocidade com que os dados trafegam pela rede. Há franquias que possuem um limite de dados transmitidos no mês e muitas vezes cancelam o acesso à internet quando este limite é atingido ou diminuem drasticamente a quantidade de dados enviados por vez, tornando assim o download de arquivos ou páginas mais lento.

A técnica de minificação

A ideia é diminuirmos ao máximo o tamanho dos nossos arquivos, algo que já fizemos com imagens. Vamos abrir o script `projeto/src/js/ativa-filtro.js` . Ele possui a seguinte estrutura:

```
$('[name=tamanho]').on('input', function(){
    $('[name=valortamanho]').text(this.value);
});
```

Veja que temos quebra de linha e todo nosso código é indentado para facilitar a legibilidade e manutenção. Porém, toda essa questão de organização é para seres humanos, o navegador consegue perfeitamente entender o código como:

```
$('[name=tamanho]').on('input', function(){$('[name=valortamanho]').text(this.value);});
```

Neste exemplo, removemos pulos de linha, inclusive colocamos todo o código `inline` , isto é, em uma única linha. Pode parecer pouco, mas se fizermos isso em todos os scripts do nosso projeto conseguiremos poupar alguns bytes que no cômputo total farão diferença para o usuário final, onerando menos sua largura de banda. Essa técnica se chama **minificação**.

É claro que todo esse processo não pode ser feito com os arquivos originais nem mesmo manualmente. Imagine que para cada novo arquivo ou para cada arquivo alterado termos que realizar esse processo. Para essa finalidade, existe o plugin gulp-uglify (https://www.npmjs.com/package/gulp-uglify). Ele é baixado como qualquer outro módulo do Node.js:

```
npm install gulp-uglify@1.4.1 --save-dev
```

Não é novidade, precisamos realizar o require do módulo em nosso `gulpfile.js` :

```
// gulpfile.js
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin')
  ,clean = require('gulp-clean')
  ,concat = require('gulp-concat')
  ,htmlReplace = require('gulp-html-replace')
```

```
  ,uglify = require('gulp-uglify');
// código posterior omitido
```

E agora? Como utilizá-lo? Quais arquivos ele deve minificar? Bem, vamos dar uma olhada na tarefa `build-js` :

```
gulp.task('build-js', function() {
    return gulp.src(['dist/js/jquery.js',
    'dist/js/home.js',
    'dist/js/ativa-filtro.js'])
  .pipe(concat('all.js'))
  .pipe(gulp.dest('dist/js'));
});
```

Temos stream de leitura que passa pelo stream do `concat` e o resultado final, o arquivo concatenado, é salvo na pasta correta. Vamos fazer com que o resultado da concatenação seja processado pelo `uglify` ? Para isso, vamos encadear uma chamada à função `pipe` :

```
gulp.task('build-js', function() {
    return gulp.src(['dist/js/jquery.js',
    'dist/js/home.js',
    'dist/js/ativa-filtro.js'])
  .pipe(concat('all.js'))
  .pipe(uglify())
  .pipe(gulp.dest('dist/js'));
});
```

Como estamos trabalhando com streams, o processo de concatenação e minificação serão feitos em memória, isto é, o resultado final é gravado de uma só vez no disco. A vantagem disso é que nossa tarefa será processada muito rapidamente já que não terá que fazer vários acesso ao disco.

Agora, precisamos rodar no terminal e verificar o resultado:

```
npm run gulp
```

Vamos abrir `projeto/dist/js/all.js` . Nele temos todos os arquivos concatenado, inline. Inclusive o processo de minificação encurta nome de variáveis para tornar ainda menor nosso arquivo (claro, modifica os nomes e troca em todos os lugares que são chamados):

```
function trocaBanner(){document.querySelector(".destaque img").src=banners[0],banners.reverse()
```

Apesar de tudo funcionar, precisamos levar alguns pontos em consideração. Nem sempre é ideal juntarmos todos os arquivo em um só, queremos juntar apenas os scripts que pertencem à página. O motivo disso muitas vezes é que gerar um arquivo "gigantão" resolve o problema de latência, mas acaba comprometendo a largura de banda. Deve haver equilíbrio.

Mas imagine configurar nosso `gulpfile.js` para que tenha uma lista de arquivos que serão concatenados por página? E não podemos nos esquecer de colocá-los na ordem correta. Trabalhoso não?

Foi pensando nesse problema que foi criado o plugin gulp-usemin (https://github.com/zont/gulp-usemin). O primeiro passo é instalá-lo no terminal e importá-lo em nosso `gulpfile.js`:

```
npm install gulp-usemin@0.3.14 --save-dev
```

```
// gulpfile.js
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin')
  ,clean = require('gulp-clean')
  ,concat = require('gulp-concat')
  ,htmlReplace = require('gulp-html-replace')
  ,uglify = require('gulp-uglify')
  ,usemin = require('gulp-usemin');
```

Por incrível que pareça, não precisamos mais das tarefas `build-html` nem `build-js`. Podemos apagá-las no nosso `gulpfile.js`.

```
// REMOVER essa tarefa do gulpfile.js
gulp.task('build-js', function() {
    return gulp.src(['dist/js/jquery.js',
    'dist/js/home.js',
    'dist/js/ativa-filtro.js'])
  .pipe(concat('all.js'))
  .pipe(uglify()).pipe(gulp.dest('dist/js'));
});

// REMOVER essa tarefa do gulpfile.js
gulp.task('build-html', function() {
  return gulp.src('dist/**/*.html')
    .pipe(htmlReplace({
            'js': 'js/all.js'
    }))
    .pipe(gulp.dest('dist/'));
});
```

Agora, vamos criar uma nova tarefa chamada `usemin`. Nela, teremos no stream de leitura apenas os arquivos HTML. Em seguida, ligaremos o nosso stream com `usemin`. Ele recebe como parâmetro um objeto cuja propriedade `js` contém um array com a lista de plugins que queremos aplicar. Em nosso exemplo, temos apenas um, o `uglify`:

```
gulp.task('usemin', function() {
  return gulp.src('dist/**/*.html')
    .pipe(usemin({
      js: [uglify]
    }))
    .pipe(gulp.dest('dist'));
});
```

Mas espere um pouco: sabemos que o stream de leitura considerará apenas arquivos HTML, porém no stream de escrita precisa existir os arquivos concatenados por página, inclusive o HTML precisará ser modificado para apontar para os novos arquivos. Como o usemin saberá quais arquivos juntar e onde gravar? Que mágica é essa?

Não é mágica, o `usemin` também usa comentários para adicionar meta informações em nossas páginas, algo que já fizemos para atender o `htmlReplace`. A diferença é que no próprio comentário já indicamos o nome do arquivo que será o resultado da concatenação. Vamos começar alterando `index.html`:

index.html

```
<!-- build:css css/index.min.css -->
<link rel="stylesheet" href="css/reset.css">
<link rel="stylesheet" href="css/estilos.css">
<link rel="stylesheet" href="css/mobile.css">
<!-- endbuild -->


<!-- build:js js/index.min.js -->
<script src="js/jquery.js"></script>
<script src="js/home.js"></script>
<!-- endbuild -->
```

Usamos o mesmo comentário, inclusive `build:css` para o grupo CSS e `build:js` para o grupo JS. A diferença é que no comentário indicamos o nome do arquivo que será criado. O sufixo `.min` não foi coincidência, é para indicar que o arquivo estará minificado.

Será que está tudo certo? Precisamos testar, mas primeiro vamos alterar a tarefa `default` para executar nossa tarefa `usemin`, e claro, remover o `build-html` que não existe mais. Ela ficará assim:

```
// gulpfile.js

// código anterior omitido
gulp.task('default', ['copy'], function() {
  gulp.start('build-img', 'usemin');
});
// código posterior omitido
```

Agora, executando no terminal:

```
npm run gulp
```

Será? O arquivo `dist/js/index.min.js` foi criado e seu conteúdo é a concatenação e minificação de todos os scripts da página. Excelente! Abrindo `index.html`, vemos também que não temos mais duas tag's scripts, apenas uma.

Será que funcionou para os CSS's? Bem, o HTML também só está apontado para um único arquivo, inclusive o arquivo `projeto/dist/css/index.min.css` existe. Abrindo o arquivo, temos uma surpresa: ele não está minificado. Isso acontece porque não indicamos para o `usemin` qual plugin ele deve utilizar para minificação de arquivos CSS. Vamos instalar o gulp-cssmin (https://www.npmjs.com/package/gulp-cssmin):

```
npm install gulp-cssmin@0.1.7 --save-dev
```

Vamos importá-lo e adicioná-lo na configuração do `usemin`:

```
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin')
  ,clean = require('gulp-clean')
  ,concat = require('gulp-concat')
  ,htmlReplace = require('gulp-html-replace')
  ,uglify = require('gulp-uglify')
  ,usemin = require('gulp-usemin')
  ,cssmin = require('gulp-cssmin');

// código omitido

gulp.task('usemin', function() {
  return gulp.src('dist/**/*.html')
    .pipe(usemin({
      js: [uglify],
      css: [cssmin]
    }))
    .pipe(gulp.dest('dist'));
});
```

Agora é só rodar e curtir o resultado:

```
npm run gulp
```

Os arquivos CSS's foram concatenados e minificados!

Gostou de como o `usemin` nos ajuda nesta tarefa de concatenação e minificação de scripts? Nosso script ficou bem mais simples. Agora, se quisermos que as outras páginas também tenham seus arquivos concatenados e minificados é só adicionar o comentário correto em seus arquivos, mas isso é uma tarefa para os exercícios.