GULP

Workflow e sua automação

Gulp

Se você é um desenvolvedor front-end já deve ter se preocupado não apenas em otimizar imagens, mas em minificar e concatenar seus scripts, inclusive já deve ter experimentado o uso de algum pré-processador como LESS.

A qualidade e quantidade das tarefas variam de acordo com as necessidades do seu projeto e essas tarefas acabam gerando um fluxo (workflow) identificável e que muitas vezes é documentado para ser usado por toda a equipe.

O problema é que tudo que é feito pelo ser humano está sujeito a erro. Por mais que tenhamos um manual nos dizendo o que fazer, nada nos impede de pularmos um dos passos, o que pode afetar diretamente o resultado final.

Para solucionar problemas como esse, foram criadas no mercado ferramentas de construção (build) de projetos como Ant, Gradle e Maven, mas há aquelas que são voltadas especialmente para programadores front-end como Grunt e o Gulp.

Gulp

O Gulp é uma ferramenta de build totalmente feita em JavaScript tornando-a atrativa no mundo front-end, especialmente para nosso projeto. Porém, para que funcione, precisamos do Node.js instalado em nosso máquina.

A primeira coisa é termos o Gulp instalado para que possamos criar nossos scripts de automação. Sua instalação é feita através do terminal e dentro da pasta projeto

O Gulp é um módulo do Node.js e todos os seus módulos são instalados através do npm. O npm acessa um repositório público na web com vários projetos que podem ser consumidos pela nossa aplicação, inclusive o Gulp.

Agora vamos baixar o Gulp através do comando npm install. Este comando recebe como parâmetro o nome do módulo, em nosso caso, gulp. Por fim, ainda queremos que o módulo fique listado em package.jsonatravés de --save-dev:

```
npm install gulp --save-dev
```

O arquivo gulpfile.js

O primeiro passo, antes de qualquer tarefa, é termos acesso ao Gulp que baixamos através do npm em nosso gulfile.js. Quando queremos carregar um módulo em nosso script usamos a função require que recebe como parâmetro o nome do módulo.

```
var gulp = require('gulp');
```

É através da variável gulp que interagimos com um objeto que representa o módulo Gulp. Este objeto possui uma série de métodos auxiliares. Qual deles usaremos? Com certeza aquele que lê a pasta com as imagens do nosso projeto, os arquivos origem da nossa tarefa. É por isso que existe o método gulp.src que recebe como parâmetro o nome de um arquivo ou de uma pasta. Alterando nosso script:

Lint e detecção de erros em JavaScript

Existe um plugin do gulp chamado gulp-jshint, que analisa nosso arquivo JS e nos dá dicas (hints) de problemas ou melhorias que podem ser feitas. Na computação, existe um termo para esse procedimento que procura alguma estrutura suspeita ou não conformante com determinada sintaxe, esse procedimento se chama lint.

```
npm install gulp-jshint --save-dev
var jshint = require('gulp-jshint');
var gulp  = require('gulp');

gulp.task('lint', function() {
  return gulp.src('./lib/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'));
});
```

```
// gulpfile.js
var gulp = require('gulp');
gulp.src('public/img');
gulp.dest('public/img');
```

O que esse comando faz é gerar um fluxo de leitura para a origem projeto/public/img e todos os seus arquivos. A mesma, pois queremos ler as imagens, otimizá-las e gravá-las em seu local de origem. Definimos o destino através do método gulp.dest que cria um fluxo de escrita para a mesma pasta.

Temos um fluxo de leitura e nosso fluxo de escrita: leremos as imagens e gravaremos no mesmo lugar, Onde fica a minificação das imagens? Teria que ser realizada entre o fluxo de leitura e o de escrita Mas o Gulp não vem com esse recurso por padrão. Para essa usaremos o plugin do Gulp chamado gulp-imagemin. Todo plugin do Gulp é também um módulo do Node.js e deve ser instalado através do npm. `npm install gulp-imagemin --save-dev`

Agora, em nosso gulpfile.js, vamos importar o módulo através da função require em uma variável chamada imagemin:

```
// gulpfile.js
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin');
gulp.src('public/img');
gulp.dest('public/img');
```

agora precisamos saber como conectar o fluxo de leitura ao imagemin que precisa saber quais arquivos considerar e este último ao fluxo de escrita para gravar os arquivos resultantes da minificação.

Usamos do método .pipe (tubo) que ligamos fluxos, seja ele de leitura ou escrita. Ele recebe como parâmetro outro fluxo que desejamos nos conectar. Vamos ligar o fluxo de leitura ao imagemin e este último ao fluxo de escrita:

```
gulp.src('public/img').pipe(imagemin()).pipe(gulp.dest('public/img'));
```

Criando Tarefas

Configuramos uma tarefa, mas não é nossa tarefa padrão, muito menos tem um nome. No Gulp, precisamos executar nosso código através de tarefas e toda tarefa possui um nome. É por isso que usamos a funçãogulp.task para criar tarefas. Nossa tarefa se chamará build-img

```
gulp.task('build-img', function() {
  gulp.src('public/img/**/*')
    .pipe(imagemin())
    .pipe(gulp.dest('public/img'));
});
```

O método gulp.task recebe como primeiro parâmetro o nome da nossa tarefa e como segundo uma função que será executada assim que a tarefa foi chamada através do Gulp no terminal.

Automatizando a concatenação merge

Vamos realizar a concatenação automática de scripts através do plugin gulp-concat. Instalando-o no terminal com o npm:

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

```
// importou gulp-concat
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin')
  ,clean = require('gulp-clean')
  ,concat = require('gulp-concat');
gulp.task('build-js', function() {
         gulp.src('dist/js/**/*.js')
        .pipe(concat('all.js'))
        .pipe(gulp.dest('dist/js'));
});
```

O arquivo dist/js/all.js foi criado, porém ainda temos um problema. Se abrirmos a páginadist/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.

```
npm install gulp-html-replace --save-dev
htmlReplace = require('gulp-html-replace');
```

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

```
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.

```
<!-- 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:

```
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 aqui, lá no comentário usaríamos build:aqui. Veja que seu valor é o nome do arquivo que substituirá o bloco do comentário no HTML.

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-htmlquanto 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:

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

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:

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

O problema de grandes arquivos

A técnica de minificação

A ideia é diminuirmos ao máximo o tamanho dos nossos arquivos, algo que já fizemos com imagens. Vamos abrir um scrip. Ele possui a seguinte estrutura:

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 final farão diferença para o usuário final. 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.

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

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 várias acesso ao disco.

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.

Foi pensando nesse problema que foi criado o plugin gulp-usemin.

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'));
});
```

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?

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:

```
<!-- 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 -->
```

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.

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

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

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, 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:

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

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