02

Integridade do projeto

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/gulp/stages/02-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.

Processos destrutivos

Muito bem! Aprendemos a automatizar nossa primeira tarefa, a minificação de imagens. Que tal rodarmos nossa tarefa mais uma vez para vermos se tudo continua funcionando? No terminal:

```
npm run gulp build-img
```

Nenhum mensagem de erro é exibida, porém não houve minificação alguma:

```
Starting 'build-img'...
Finished 'build-img' after 10 ms
gulp-imagemin: Minified 32 images (saved 0 B - 0%)
```

Não houve minificação porque as imagens já foram minificadas quando rodamos pela primeira vez nossa tarefa. Mistério resolvido!

Agora, antes de passarmos para a próxima tarefa, vou "dar uma de design" e retocarei o banner `projeto/img/destaque-home.png` abrindo-a no editor de imagens padrão da nossa plataforma. Vamos deixá-la mais clara:

Pronto, agora só rodarmos nossa tarefa mais uma vez:

```
npm run gulp build-img
```

Vejamos o resultado no console:

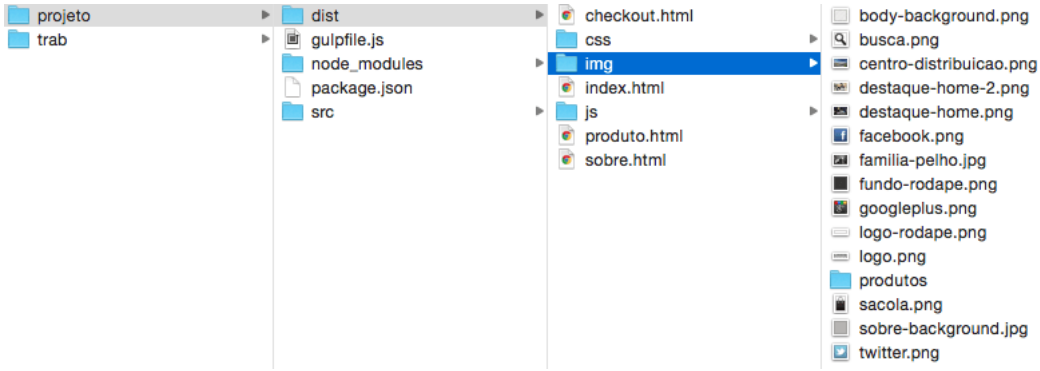
```
[11:40:15] Starting 'build-img'...
[11:40:15] Finished 'build-img' after 12 ms
[11:40:19] gulp-imagemin: Minified 32 images (saved 149.98 kB - 7.3%)
```

Curioso! Quando alteremos nossa imagem pelo editor de imagem o resultado final é uma imagem não otimizada, por isso nossa tarefa a otimizou. Fizemos uma simples alteração na imagem, mas se repetirmos esse processo diversas vezes? Cada vez mais a imagem perderá sua qualidade porque estaremos sempre trabalhando com uma imagem já otimizada. O problema aqui é a destrutividade do nosso processo, que modifica os arquivos originais.

Lindando com processos destrutivos

Uma maneira de resolver esse problema é guardando as imagens originais. Sempre que uma alteração for necessária elas serão utilizadas e o processo de minificação operará sobre uma cópia. A mesma lógica se aplica com todos os outros arquivos do projeto.

Olhando nosso projeto, temos a pasta `src`. Vamos criar uma cópia desta pasta que chamaremos de `dist`. Essa pasta, a famosa pasta `distribuição` é aquela que será colocada no ar, sendo assim, qualquer otimização que fizermos seja de imagens ou qualquer outro arquivo será nos arquivos desta pasta e nunca nos originais.



Agora, vamos alterar nossa tarefa `build-img` para considerar a pasta `dist`, preservando a pasta `src`:

```
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin');

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

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

Pronto! Quando rodarmos nossa task `build-img` no terminal, estaremos trabalhando com uma cópia da pasta `src`, preservando assim as imagens originais e qualquer outro arquivo que viermos a alterar.

Espere um pouco! E se rodarmos nossa tarefa novamente? Teremos que apagar a pasta `dist` e copiar novamente a pasta `src`. Mas não é só isso, se adicionarmos mais uma imagem em `src/img` teremos que lembrar de copiar essa imagem para a pasta `dist`. Repare que estamos realizando todo esse processo manualmente e não é isso que queremos, certo? Vamos automatizar a tarefa que copia o conteúdo da pasta `src` para a pasta `dist`.

Automatizando a cópia de arquivos

Já aprendemos a criar um fluxo de leitura (origem) e um fluxo de escrita (destino). Que tal ligarmos um fluxo no outro diretamente? O resultado disso será uma cópia do conteúdo do fluxo de leitura para o fluxo de escrita, justamente o que precisamos. É claro, precisamos criar uma tarefa para isso que vamos chamar de `copy`:

```
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin');

// nova tarefa
gulp.task('copy', function() {
    gulp.src('src')
    .pipe(gulp.dest('dist'));
});

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

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

Tarefa criada, agora vamos testar no console:

```
npm run gulp copy
```

Funcionou! Recebemos a seguinte mensagem no terminal:

```
Starting 'copy'...
Finished 'copy' after 9.17 ms
```

A pasta `dist` foi criada, porém temos um problema. O seu conteúdo deveria ser o conteúdo da pasta `src`, mas o que a pasta `dist` contém é a pasta `src`. Para resolvermos isso, precisamos alterar nosso fluxo de leitura para copiar o conteúdo da pasta `projeto/src` e não a pasta em si:

```
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin');

// alterando nossa task
gulp.task('copy', function() {
    gulp.src('src/**/*')
        .pipe(gulp.dest('dist'));
});

gulp.task('build-img', function() {
```

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

Usamos o padrão `**` para indicarmos que queremos copiar todas as pastas e `*` para indicar que queremos todos os arquivos dessa pasta. Isso é o que chamamos de **glob pattern**. Vamos testar mais uma vez:

```
npm run gulp copy
```

Excelente, ele copiou todo o conteúdo da pasta `projeto/src` para dentro de `dist` .

Precisamos apagar antes de copiar

A pasta `src` que havia sido copiada indevidamente ainda continua lá. Podemos resolver isso facilmente apagando a pasta `projeto/dist/src` , mas será isso a melhor solução?

Para complicar ainda mais, vamos apagar uma imagem do projeto original. Não é incomum adicionarmos e removermos imagens. Se executarmos nossa task `copy` ele copiará todo o conteúdo de `src` , perfeito, mas como não apagamos a pasta `dist` antes de copiar, a imagem apagada na pasta original ainda estará em `dist` .

Nesse caso, quem deve ser o nosso fluxo de origem é a pasta `projeto\dist` , mas não queremos simplesmente gravá-la em outro lugar através de um fluxo de escrita, queremos é apagá-la. Para essa tarefa não rotineira, existe um plugin do Gulp com essa finalidade o gulp-clean (https://github.com/peter-vilja/gulp-clean).

Já aprendemos que é através do do npm executado no terminal que instalamos plugins do Gulp, que nada mais são do que módulos do Node.js. Vamos lá? Em nosso terminal:

```
npm install gulp-clean@0.3.1 --save-dev
```

Como já vimos, a instalação desse módulo criará a pasta `node_modules/gulp-clean` e como usamos `--save-dev` ele será adicionado como dependência em nosso arquivo `package.json` .

Agora, em nosso `gulpfile.js` , vamos importar o módulo através da função `require` e para nossa comodidade armazená-lo em uma variável de nome curto chamada `clean` :

```
// importando o módulo del
var gulp = require('gulp')
  ,imagemin = require('gulp-imagemin')
  ,clean = require('gulp-clean');

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

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

  gulp.src('dist/img/**/*')
```

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

Excelente. E agora? A tarefa de copiar é diferente da tarefa de apagar é por isso que criaremos uma nossa tarefa chamada `clean`:

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


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

// a nova tarefa clean
gulp.task('clean', function() {
    gulp.src('dist')
        .pipe(clean());
});

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

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

Será que funciona? Vamos rodar primeiro a tarefa `clean` e logo em seguida a tarefa `copy`:

```
npm run gulp clean
npm run gulp copy
```

Perfeito! Funcionou! Primeiro ele apaga e depois copia.

Tarefas e suas dependências

Nosso treinamento é de automação de tarefas e sabemos que tudo que é feito pelo ser humano esta sujeito a erros. Certo? E se alguém esquecer de rodar a tarefa clean antes da copy, ou pior, inverter a ordem? Será que há alguma maneira de executarmos as duas tarefas de uma só vez? Sim.

Podemos indicar em uma tarefa qual é sua dependência, isto é, qual tarefa deve ser executada antes. Vamos alterar nossa task `copy` que depende da task `clean`:

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

// adicionando clean como dependência da tarefa copy
```

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

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

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

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

Veja que antes da função de callback, passamos um array, uma lista que contém todas as tarefas que desejamos executar antes. Será que funciona? Vamos agora executar apenas a task **copy**:

```
npm run gulp copy
```

Não funcionou e ainda exibiu o seguinte erro:

```
events.js:85
      throw er; // Unhandled 'error' event
            ^
Error: Unable to delete "/Users/flavioalmeida/Desktop/projeto/dist" file (ENOTEMPTY, rmdir '/Us
```

O que está acontecendo? A tarefa `copy` não esperou a tarefa `clean` terminar, mesmo adicionando-a como dependência. Para resolvermos esse problema, vamos lançar mão de um termo mais apropriado para a palavra fluxo, usaremos o termo **stream**. Quando combinamos streams, o resultado final é um novo stream. Por exemplo, poderíamos ter guardado o stream final em uma variável:

```
// não entra em nenhum lugar, apenas um exemplo
gulp.task('clean', function() {
    var stream = gulp.src('dist')
        .pipe(clean());
});
```

Se não retornarmos um stream, então o resultado assíncrono de cada tarefa não será aguardado por quem a chamou, inclusive por nenhuma de suas tarefas dependentes. Retornando uma stream:

```
// não entra em nenhum lugar, apenas um exemplo
gulp.task('clean', function() {
    var stream = gulp.src('dist')
        .pipe(clean());
    return stream;
});
```

Bom, agora vamos alterar pra valer nosso código, mas retornando o stream diretamente, sem guardar em uma variável:

```
// gulpfile.js
// código anterior omitido
gulp.task('clean', function() {
    return gulp.src('dist')
        .pipe(clean());
});
// código posterior omitido
```

Vamos testar para ver se realmente a tarefa `copy` aguardou a tarefa `clean` terminar?

```
npm run gulp copy
```

Funcionou! A tarefa copy só foi executada depois da conclusão da tarefa clean.

```
Starting 'clean'...
Finished 'clean' after 35 ms
Starting 'copy'...
Finished 'copy' after 4.35 ms
```

Excelente, agora apenas um comando no terminal apaga e copia os nossos arquivos.

Então, que tal fazermos todo o processo? Copiar, apagar e otimizar as imagens? Opa! Mas se ele tentar otimizar as imagens antes de copiar nossos arquivos? Que tal adicionarmos `build-img` também como dependência de `copy` ?

```
// código anterior omitido
gulp.task('copy', ['clean', 'build-img'], function() {
    gulp.src('src/**/*')
        .pipe(gulp.dest('dist'));
});
// código posterior omitido
```

Temos um erro!

```
  throw er; // Unhandled 'error' event
```

Veja que temos um problema aqui. A tarefa `build-img` só deve ser processada quando `copy` e todas as suas dependências forem executadas. Cometemos um erro colocando `build-img` como dependência de `copy` , é o contrário! Vamos alterar nosso script. Inclusive, vamos retornar o stream da tarefa copy, porque a tarefa `build-img` só pode ser executada depois da tarefa `copy` completar:

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

// removida a dependência de build-img
gulp.task('copy', ['clean'], function() {
```

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

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

// adicionando a dependência copy
gulp.task('build-img', ['copy'], function() {

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

Agora é só testar e verificar o resultado:

```
npm run gulp build-img
```

Perfeito, as tarefas `clean` , `copy` e `build-img` são executadas. Veja que todas as tarefas são executadas uma por vez, porque nesse contexto não faz sentido que sejam executadas em paralelo (`build-img` depende de `copy` que depende de `clean`) Ainda recebemos a mensagem de 0% de minificação de imagens, mas isso era esperado. Precisamos recuperar os arquivos originais do projeto que foram modificados anteriormente. Volte ao projeto do capítulo anterior e copie as imagens e teste novamente.

Chegou a hora dos exercícios!