

O canivete suíço
do desenvolvedor

Node

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2016]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 - Vila Mariana - São Paulo - SP - Brasil

www.casadocodigo.com.br

AGRADECIMENTOS

Gostaria primeiramente de agradecer ao Adriano, por ter acreditado no livro, sugerido o tema e, principalmente, por ter acreditado no garoto do interior que, há alguns anos atrás, teve seu primeiro emprego como estagiário no protótipo da Editora em que hoje escrevo.

Gostaria de agradecer também as pessoas que trabalham comigo pelo constante incentivo à aprendizagem e melhoria, principalmente aos meus amigos Maurício Aniche e Guilherme Silveira.

Por fim, mas não menos importante, gostaria de agradecer aos meus pais, por sempre estarem ao meu lado e também me incentivarem a sempre dar o melhor de mim.

SOBRE O AUTOR

Meu nome é Caio Incau, e trabalho com desenvolvimento de software. Comecei aos 16 anos estudando por conta em casa, na época com Delphi.

Aos 17 anos, entrei na faculdade para cursar Sistemas de Informação. Também nessa idade, tomei uma das melhores decisões que já tive: decidi estudar Java pela Caelum. Com 18 anos, comecei a trabalhar na Caelum, empresa onde trabalho até o presente momento.

Durante minha estadia no mercado de TI, tive a oportunidade de trabalhar com Java, Ruby, JavaScript e Objective-C.

Busco sempre me atualizar e aprender sobre novas tecnologias, pois acredito fortemente que este é o segredo para o sucesso em nossa área de trabalho.

Sumário

1	Introdução	1
1.1	Para quem é este livro	1
1.2	Como este livro funciona	2
1.3	Configurando o projeto de exemplo na sua máquina	2
2	Melhorando a performance do lado do cliente	7
2.1	Resolvendo o problema	8
2.2	Revisando	25
3	Cuidando de erros e logs	26
3.1	Trabalhando com os erros HTTP	26
3.2	Um pouco mais sobre o objeto Request do Express	31
3.3	Um pouco mais sobre o objeto Response do Express	34
3.4	Mantendo histórico de nossa aplicação	38
3.5	Ambientes de desenvolvimento	43
3.6	Reduzindo a quantidade de erros com express-validator	46
3.7	Revisando	54
4	Melhorando performance e segurança	55
4.1	GZIP	55
4.2	Usando cache de navegador	59
4.3	Favicon	61

4.4 Node e Threads	62
4.5 Uma opção ao Cluster nativo	66
4.6 Protegendo a aplicação	69
4.7 Revisando	75
5 Envio de e-mails com Node.js	77
5.1 Mantendo sua aplicação de pé	81
5.2 Migrando do Express 3 para o 4	84

INTRODUÇÃO

Você já passou por problemas reais ao fazer o deploy de uma aplicação com Express? Como, por exemplo, servir conteúdo estático de forma otimizada, tratar os erros, usar cache, ou até mesmo a necessidade de criar um Cluster?

Pois bem, é isto que este livro aborda: como resolver ou se prevenir de problemas na sua aplicação utilizando Express.

A ideia é tratar de diversos conceitos básicos e avançados que resolvam alguns dos principais impasses do dia a dia de um desenvolvedor.

1.1 PARA QUEM É ESTE LIVRO

Este livro é para as pessoas que já possuem algum conhecimento em NodeJS, Express e Mongoose. Não é necessária uma grande experiência com essas ferramentas, apenas o básico.

Você pode seguir o livro sem o conhecimento prévio delas, porém será mais complicado de entender o código de exemplo.

Todo o código estará no meu repositório do GitHub, que você pode acessar em: <https://github.com/CaioIncau/my-todo/tree/cap1>, dividido por capítulos. Espero que você aproveite o que aprender aqui, e aplique em seus projetos, sejam eles pessoais ou empresariais.

Existe um grupo de discussão deste livro, no qual você pode postar dúvidas e compartilhar seus resultados: <http://forum.casadocodigo.com.br/>.

1.2 COMO ESTE LIVRO FUNCIONA

Este livro vai usar um projeto extremamente simples como exemplo. Será uma aplicação na qual você cadastra tarefas e, por isso, será chamada de *Todo*.

A proposta é que você veja o conceito e o transporte para o seu projeto, ou até mesmo que aplique diretamente nele: o *Todo* é apenas um apoio didático.

A cada capítulo, vamos avançando mais o projeto e adicionando novas funcionalidades voltadas para segurança, performance e manutenibilidade.

1.3 CONFIGURANDO O PROJETO DE EXEMPLO NA SUA MÁQUINA

O projeto de exemplo está no GitHub, e ele pode ser clonado em seu estado inicial, neste repositório: <https://github.com/CaioIncau/my-todo/tree/inicial>.

Se você não é familiarizado com o Git, você pode baixar a versão zipada em: <https://github.com/CaioIncau/my-todo/archive/inicial.zip>.

Se você não tem o Node instalado, lembre-se de baixá-lo em <https://nodejs.org/download/>.

Rode o comando `node -v` e garanta que seu Node utiliza a versão 0.10 ou superior:


```
caioincau@MBA/my-todo-inicial: (master)$ node -v
v0.10.25
```

Figura 1.1: Node version

Após baixar o projeto e deszipá-lo, entre na pasta `my-todo` e rode o comando `npm install`. Usaremos o `npm` para resolver as dependências do projeto. Se precisar, delete a pasta `.node_modules` antes de realizar o `npm install`, pois pode haver conflitos com versões preexistentes dos módulos.

A saída do `npm install` será algo próximo a isto:

```
caioincau@MBA/my-todo-inicial: (master)$ npm install
npm WARN package.json my-todo@1.0.0 No repository field.
npm WARN package.json my-todo@1.0.0 No README data
npm WARN deprecated mongoose@3.8.17: Bad bug with save() - see git
hub issue #2340

> kerberos@0.0.3 install /Users/caioincau/Documents/my-todo-inicia
l/node_modules/mongoose/node_modules/mongodb/node_modules/kerberos
> (node-gyp rebuild 2> builderror.log) || (exit 0)

CXX(target) Release/obj.target/kerberos/lib/kerberos.o
CXX(target) Release/obj.target/kerberos/lib/worker.o
CC(target) Release/obj.target/kerberos/lib/kerberosgss.o
CC(target) Release/obj.target/kerberos/lib/base64.o
CXX(target) Release/obj.target/kerberos/lib/kerberos_context.o
SOLINK_MODULE(target) Release/kerberos.node
SOLINK_MODULE(target) Release/kerberos.node: Finished

> bson@0.2.12 install /Users/caioincau/Documents/my-todo-inicial/n
ode_modules/mongoose/node_modules/mongodb/node_modules/bson
> (node-gyp rebuild 2> builderror.log) || (exit 0)

CXX(target) Release/obj.target/bson/ext/bson.o
SOLINK_MODULE(target) Release/bson.node
SOLINK_MODULE(target) Release/bson.node: Finished
ejs@1.0.0 node_modules/ejs

cookie-parser@1.3.3 node_modules/cookie-parser
├── cookie@0.1.2
└── cookie-signature@1.0.5

method-override@2.2.0 node_modules/method-override
└── vary@1.0.0
```

```

├─ parseurl@1.3.0
├─ methods@1.1.0
├─ debug@2.0.0 (ms@0.6.2)

ejs-locals@1.0.2 node_modules/ejs-locals
├─ ejs@0.8.8

serve-static@1.9.2 node_modules/serve-static
├─ utils-merge@1.0.0
├─ escape-html@1.0.1
├─ parseurl@1.3.0
├─ send@0.12.2 (destroy@1.0.3, fresh@0.2.4, ms@0.7.0, range-parse
@1.0.2, depd@1.0.0, debug@2.1.3, mime@1.3.4, on-finished@2.2.0, e
tag@1.5.1)

body-parser@1.9.0 node_modules/body-parser
├─ media-typer@0.3.0
├─ bytes@1.0.0
├─ raw-body@1.3.0
├─ depd@1.0.0
├─ qs@2.2.4
├─ iconv-lite@0.4.4
├─ on-finished@2.1.0 (ee-first@1.0.5)
├─ type-is@1.5.7 (mime-types@2.0.10)

express@4.9.5 node_modules/express
├─ utils-merge@1.0.0
├─ cookie@0.1.2
├─ fresh@0.2.4
├─ merge-descriptors@0.0.2
├─ escape-html@1.0.1
├─ cookie-signature@1.0.5
├─ finalhandler@0.2.0
├─ range-parser@1.0.2
├─ vary@1.0.0
├─ media-typer@0.3.0
├─ methods@1.1.0
├─ parseurl@1.3.0
├─ serve-static@1.6.5
├─ depd@0.4.5
├─ path-to-regexp@0.1.3
├─ qs@2.2.4
├─ debug@2.0.0 (ms@0.6.2)
├─ on-finished@2.1.1 (ee-first@1.1.0)
├─ etag@1.4.0 (crc@3.0.0)
├─ proxy-addr@1.0.7 (forwarded@0.1.0, ipaddr.js@0.1.9)
├─ send@0.9.3 (destroy@1.0.3, ms@0.6.2, on-finished@2.1.0, mime@1
.2.11)

```

```

├── type-is@1.5.7 (mime-types@2.0.10)
├── accepts@1.1.4 (negotiator@0.4.9, mime-types@2.0.10)
mongoose@3.8.17 node_modules/mongoose
├── regexp-clone@0.0.1
├── muri@0.3.1
├── sliced@0.0.5
├── hooks@0.2.1
├── mpath@0.1.1
├── mpromise@0.4.3
├── ms@0.1.0
├── mquery@0.8.0 (debug@0.7.4)
├── mongodb@1.4.9 (readable-stream@1.0.33, kerberos@0.0.3, bson@0
2.12)

```

Ele vai baixar as dependências de forma recursiva, e nos mostrará a árvore. Todo o livro corre com as dependências nas versões definidas no código anterior.

Agora precisamos iniciar o servidor do MongoDB, para que o projeto consiga se conectar ao banco de dados.

Se você tem o Mongo instalado em seu computador, basta rodar o comando `mongod` em seu terminal e o servidor subirá. Caso não possua, você pode encontrar detalhes de como instalá-lo em cada SO na documentação oficial: <http://docs.mongodb.org/manual/installation/>.

Agora, com o projeto baixado e o Mongo rodando, basta entrar na pasta do projeto e rodar o comando `node app.js`, que vai subir a aplicação e se conectar ao Mongo, criando automaticamente o banco `my-todo`.

Acesse seu localhost na porta `3001`, e verifique que a aplicação está de pé!



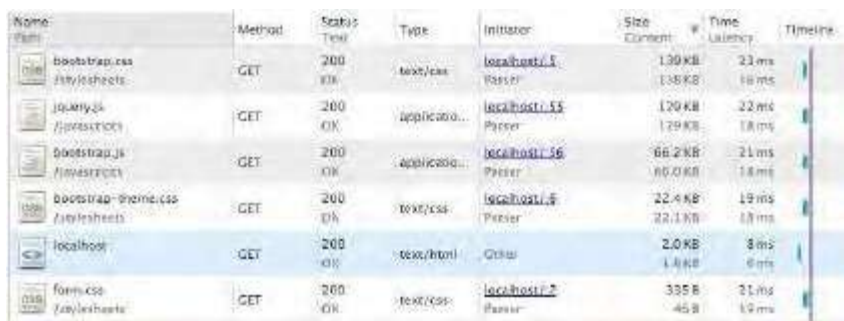
Figura 1.2: App Running

Pronto! Agora nós podemos começar a melhorar nossa aplicação que usa Node, Express e MongoDB.

MELHORANDO A PERFORMANCE DO LADO DO CLIENTE

Repare que, em nossa aplicação de exemplo, importamos apenas três arquivos de estilo (`.css`) e dois arquivos de script (`.js`). Ainda assim, podemos e devemos melhorar o modo de servi-los. Isso é algo extremamente importante, pois o tempo de resposta da página e seu consumo de dados são partes em que o usuário mais repara.

Na imagem a seguir, podemos ver quanto pesam os principais componentes de nossa página:



Nome (Path)	Method	Status Code	Type	Initiator	Size Content	#	Time Latency	Timeline
bootstrap.css	GET	200	text/css	localhost:55	130 KB		23 ms	
stylesheets	GET	OK		Parser	130 KB		10 ms	
jquery.js	GET	200	application/javascript	localhost:55	120 KB		22 ms	
bootstrap.js	GET	OK	application/javascript	Parser	129 KB		18 ms	
bootstrap.js	GET	200	application/javascript	localhost:55	66.2 KB		21 ms	
bootstrap-theme.css	GET	OK	text/css	Parser	60.0 KB		18 ms	
bootstrap-theme.css	GET	200	text/css	localhost:55	22.4 KB		19 ms	
stylesheets	GET	OK		Parser	22.1 KB		10 ms	
localhost	GET	200	text/html	Client	2.0 KB		8 ms	
form.css	GET	OK	text/css	localhost:55	1.4 KB		6 ms	
stylesheets	GET	200	text/css	localhost:55	335 B		21 ms	
stylesheets	GET	OK		Parser	45 B		19 ms	

Figura 2.1: Chrome Inspector

No total, temos 0.3mb; inaceitável para um projeto do tamanho do nosso! É aí que entra a estrela deste capítulo, o gulp!

Repare que só o JQuery pesa 129kb, e isso é bem pesado para os padrões da web hoje em dia.

Além disso, enfrentamos outro problema: os servidores consomem um tempo para resolver uma requisição, antes mesmo de entregá-la. No caso de arquivos muito pequenos, como nosso `form.css`, eles demoram mais para ter sua requisição resolvida do que para serem baixados efetivamente. Seria uma boa, então, se não tivéssemos muitos arquivos pequenos sendo entregues ao navegador.

2.1 RESOLVENDO O PROBLEMA

Temos dois problemas distintos: o primeiro é a quantidade de arquivos, e o segundo é o tamanho deles.

Vamos começar pensando em como reduzir a quantidade de arquivos.

Nós queremos que nosso site tenha todo o estilo e scripts predefinidos, mas queremos entregar menos arquivos. O modo mais simples de realizar esta tarefa é entregar um arquivo `.css` e um `.js` com todo o conteúdo de uma só vez.

Podemos copiar os conteúdos, colá-los em um arquivo e importá-lo. Essa abordagem funcionaria, mas não parece algo meio trabalhoso?

Isso faria com que, a cada alteração em nosso site, precisássemos abrir todos nossos arquivos estáticos e repetíssemos o processo. Agora, imagine que, ao longo dos anos, nosso projeto cresça e tenha algumas dezenas desse tipo de arquivo.

Sem dúvidas, é uma situação insustentável a longo e médio prazo. Entretanto, se tivéssemos uma ferramenta para automatizar

isso, não seria interessante? Iríamos nos preocupar apenas em escrever nosso código como sempre fizemos.

Pois, para isso, foram criados os `task runners`, ferramentas que automatizam tarefas fáceis, mas que são trabalhosas e consomem um tempo considerável.

gulpjs

O `gulp` é uma ferramenta de automatização de build criada utilizando NodeJS e que se aproveita do poder das `streams` para trabalhar mais rápido que seu principal concorrente, o GruntJS. Vamos começar entendendo o que são as `streams`.

`Streams` permitem que você passe dados de uma função, modifique-os e mande para outra função (geralmente funções pequenas). Desta forma, trabalhamos com os dados em memória, sendo que os arquivos são lidos e escritos apenas uma vez, o que é muito mais rápido do que trabalhar com eles puramente em disco, assim como o Grunt, que a cada parte do processo escreve e lê novamente o arquivo para o disco.

Na imagem a seguir, fiz um comparativo simples de como funcionam ambas as ferramentas funcionam:

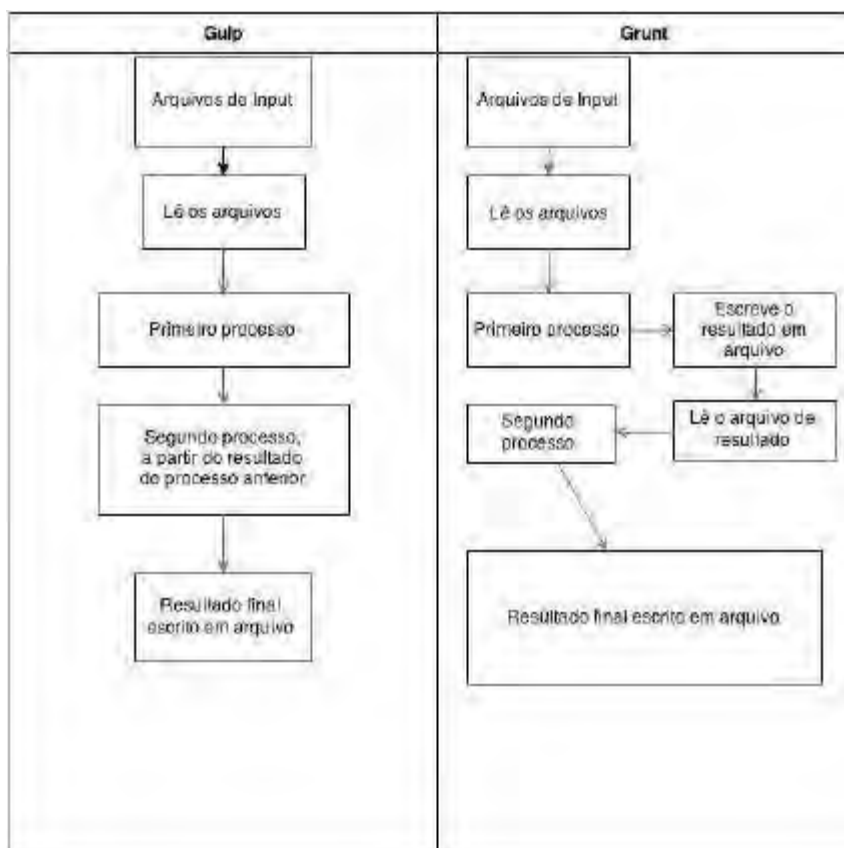


Figura 2.2: gulpjs comparator

Perceba que usamos pelo menos o dobro de I/O para realizar apenas dois processos em sequência. Agora, imagine um projeto em que temos mais de 20 ou 30 arquivos de estilo, com um encadeamento de uma dezena de funções. Quanto tempo não estaríamos ganhando ao tirar proveito das streams? Visto como a ferramenta trabalha e sua vantagem em relação à sua principal concorrente, vamos colocar a mão na massa!

Se você está lendo este livro, provavelmente já tem familiaridade com o NPM, mas de todo modo, vamos fazer uma breve recapitulação.

O NPM é usado para gerenciar os módulos em Node. E, veja só, gulp é um módulo Node! Por isso usaremos o NPM para instalar o nosso task runner.

Para isto, vamos usar o comando `npm install -g gulp`. Não se esqueça do `-g`, para que este seja um módulo de uso global.

Vamos também registrar no nosso projeto que usamos gulp por meio do `npm install --save-dev gulp`.

Para garantir que ele está instalando, rode o comando `gulp -v` em seu terminal:

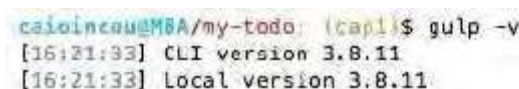
A terminal window showing the command 'gulp -v' being executed. The output displays the CLI version as 3.8.11 and the local version as 3.8.11. The prompt shows the user is in a directory named 'my-todo' and is using a shell named 'cap1'.

Figura 2.3: gulpjs comparator

Quando lidamos com outros task runners, como por exemplo o Grunt, temos o hábito de primeiro copiar todos os nossos assets para uma pasta diferente e manipular sempre essa pasta, nunca o original. Isso ocorre devido ao modo como esses task runners funcionam, reescrevendo o arquivo entre as etapas do processamento.

No gulp, nós podemos manipular diretamente os arquivos que escrevemos, sem nos preocuparmos em realizar uma cópia deles.

Trabalhando nossos arquivos CSS

Vamos começar a configurar o nosso gulp para o projeto de Todo. Para isto, criaremos na raiz do projeto um arquivo chamado `gulpfile.js`, este será a base de toda nossa configuração.

Para a maioria das tarefas (tasks) recorrentes, existem módulos (também chamados de plugins) predefinidos que nos auxiliam a

executá-las. Vamos começar pela concatenação dos nossos arquivos CSS. Com isto, reduziremos a quantidade de arquivos servidos para o nosso cliente (navegador).

Usaremos o módulo `gulp-concat`. Como o `gulp` é escrito em Node, ele pode tirar proveito da especificação de sua importação ao usar o `require`, que nada mais faz do que importar para outro arquivo `.js` as funções e variáveis que são expostas por um módulo já predefinido.

Vamos importar o nosso módulo e o próprio `gulp` no `gulpfile.js`, e instalar o módulo pelo NPM por meio do comando:

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

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

Repare no uso da flag `--save-dev`; ela será muito importante para nós no futuro. Essa flag indica para o NPM que essa dependência só é usada em desenvolvimento. Sendo assim, ele separa esse tipo de dependência em nosso `package.json`:

```
{  
  "name": "my-todo",  
  "version": "1.0.0",  
  "description": "Todo App",  
  "main": "app.js",  
  "dependencies": {  
    "body-parser": "1.9.0",  
    "cookie-parser": "1.3.3",  
    "express": "4.9.5",  
    "ejs": "1.0.0",  
    "method-override": "2.2.0",  
    "serve-static": "1.9.2",  
    "ejs-locals": "1.0.2",  
    "mongoose": "3.8.17"  
  },  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "Caio Incau",  
  "license": "ISC",  
}
```

```
"devDependencies": {
  "gulp-concat": "^2.5.2",
}
```

Para você que não está familiarizado com o `package.json`, este é o arquivo onde são encontrados os detalhes do projeto, como nome, versão, qual o arquivo principal e também suas dependências. Encontramos também informações importantes, como a licença de distribuição do software.

Após importar o módulo de concatenação, precisamos registrar a tarefa no nosso `gulpfile.js`. Para isso, usaremos o método `task` do `gulp`.

```
gulp.task('css',function(){
});
```

Repare que o método `task` recebe uma `String` e uma `function`. O primeiro parâmetro é o nome da nossa tarefa – nesse caso, chamamos de `css` –; o segundo é uma função que vai definir o conteúdo de nossa tarefa – nesse caso, vamos concatenar nosso CSS.

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

gulp.task('css', function() {
  var cssPath = {cssSrc:['./public/stylesheets/*.css', '!*.min.css', '!/**/*.min.css'], cssDest:'public'};
  return gulp.src(cssPath.cssSrc)
    .pipe(concat('styles.css'))
    .pipe(gulp.dest(cssPath.cssDest));
});
```

Vamos agora entender o que foi feito em nossa primeira tarefa. Primeiro, atribuímos em uma variável os diretórios onde estão os arquivos CSS (`cssPath`) apenas para facilitar a leitura. Trabalhamos com uma lista de arquivos, usando o `glob pattern`, e nesse array excluimos arquivos minificados para evitar conflitos futuros.

Em seguida, mostramos para o `gulp` onde os arquivos que deveriam ser lidos são encontrados. Isto é feito por meio do método `src`. Neste caso, os arquivos estão em `cssPath.cssSrc`

Logo após, chamamos o método `pipe`, que recebe uma função que processará o resultado da ação que a antecede diretamente na ordem do pipeline – nesse caso, a função `src`, que havia realizado a leitura dos arquivos. Por fim, colocamos mais uma ação na pipeline (chamando o método `pipe` novamente) e, desta vez, mostramos para o `gulp` onde deve ser escrito o arquivo com o resultado do processamento, através do método `dest`.

Se você olhar a pasta `public`, verá que tem um arquivo chamado `styles.css`, que deve conter exatamente o mesmo conteúdo que todos os arquivos CSS do projeto.

Vamos agora minificar o nosso CSS, pois ele já está concatenado. Para isto, também existe um módulo, chamado `gulp-minify-css`. Vamos também usar o `rename` para adicionar um sufixo em nosso arquivo, o `.min`, que vai indicar que esse arquivo já está minificado.

Instale ambos em nosso projeto por meio do comando `npm instal --save-dev gulp-rename gulp-minify-css` e importe em nosso `gulpfile.js`

```
var gulp = require('gulp');
var concat = require('gulp-concat');
var minifyCSS = require('gulp-minify-css');
var rename = require('gulp-rename');
// resto do código
```

Adicione este passo no pipeline da tarefa:

```
gulp.task('css', function() {
var cssPath = {cssSrc:['./public/stylesheets/*.css', '!*.min.css',
'!/**/*.min.css'], cssDest:'public'};
return gulp.src(cssPath.cssSrc)
.pipe(concat('styles.css'))
```

```

.pipe(minifyCSS())
.pipe(rename({suffix: '.min'}))
.pipe(gulp.dest(cssPath.cssDest));
});

```

Agora que já concatenamos e minificamos nossos arquivos de estilo, podemos adicionar o arquivo final em nosso `layout.ejs`, e remover a chamada dos três arquivos que usávamos antes.

Seu `layouts.ejs` deve ficar assim:

```

<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel="stylesheet" type="text/css" href="styles.min.css">
    <!--[if lt IE 9]>
      <script src="http://html5shiv.googlecode.com/svn/trunk/html5.js">
    </script>
    <![endif]-->
  </head>
  <body>
    <div class="container">
      <%- body %>
    </div>
    <script type="text/javascript" src="../javascripts/jquery.js">
  </script>
    <script type="text/javascript" src="../javascripts/bootstrap.js">
  </script>
  </body>
</html>

```

A tarefa de minificação poderia receber ainda outro parâmetro com algumas opções úteis, como `compatibility` para manter compatibilidade com versões específicas de navegadores, ou `debug` para gerar estatísticas sobre o processo de minificação. Não vamos utilizá-las aqui, mas você pode consultar todas essas opções e seus usos em <https://www.npmjs.com/package/gulp-minify-css>.

Ainda podemos dar uma última melhorada em nosso CSS ao adicionar os prefixos em propriedades não suportadas em alguns browsers, como, por exemplo, o `display: flex`. Veja que, para suportar esta propriedade do CSS3 em todos os browsers, é preciso

adicionar o prefixo relativo ao navegador:

```
display: -webkit-box;  
display: -moz-box;  
display: -ms-flexbox;  
display: flex;
```

Isso é algo que nos toma um tempo e nem sempre lembramos de colocar em todos os lugares necessários, por isso foi criado o módulo `gulp-autoprefixer`. Vamos repetir os passos para instalar, importar e adicionar na pipeline:

```
var gulp = require('gulp');  
var concat = require('gulp-concat');  
var minifyCSS = require('gulp-minify-css');  
var rename = require('gulp-rename');  
var autoprefixer = require('gulp-autoprefixer');  
  
gulp.task('css', function() {  
  var cssPath = {cssSrc:['./public/stylesheets/*.css', '!*.min.css',  
    '!/**/*.min.css'], cssDest:'public'};  
  return gulp.src(cssPath.cssSrc)  
    .pipe(concat('styles.css'))  
    .pipe(autoprefixer('last 2 versions'))  
    .pipe(minifyCSS())  
    .pipe(rename({suffix: '.min'}))  
    .pipe(gulp.dest(cssPath.cssDest));  
});
```

Neste caso, prefixamos para as duas últimas versões de cada navegador. Agora sim, temos um CSS único e completamente otimizado, reduzimos a quantidade de requisições para nosso servidor, reduzimos o tamanho das requisições, e ainda deixamos nossa aplicação mais compatível com a variedade de navegadores.

E o melhor: configurado isso uma vez, não precisamos nos preocupar em otimizar nosso CSS a cada novo arquivo ou mudança em nosso site. A vantagem de se usar ferramentas que automatizam as tarefas está justamente aí.

Trabalhando com nossos arquivos JS

Agora que já resolvemos os problemas com nossos arquivos CSS, vamos começar a tratar os nossos JavaScripts. Para isto, vamos registrar uma tarefa chamada `js`.

```
gulp.task('js', function() {  
  var jsPath = {jsSrc:['./public/javascripts/**/*.js'], jsDest:'  
  public'};  
});
```

Durante o desenvolvimento com JavaScript, costumamos a debugar nosso código através de `console.log` ou `console.error`. O problema é que por muitas vezes esquecemos de remover esses logs antes de enviarmos nossa aplicação para produção. O usuário comum dificilmente abre o console de seu navegador e verifica isso, mas, ainda assim, seria educado remover essas linhas.

Para realizar essa tarefa, nós temos o módulo `gulp-strip-debug`. Vamos instalar com `npm install --save-dev gulp-strip-debug`, e importar no nosso `gulpfile.js`

```
// resto dos requires  
var stripDebug = require('gulp-strip-debug');  
  
gulp.task('js', function() {  
  var jsPath = {jsSrc:['./public/javascripts/**/*.js'], jsDest:'  
  public'};  
  return gulp.src(jsPath.jsSrc)  
    .pipe(concat('scripts.js'))  
    .pipe(stripDebug())  
    .pipe(gulp.dest(jsPath.jsDest));  
});
```

Repare que aproveitamos para concatenar nossos arquivos JavaScript. O modo como isso foi feito é idêntico ao modo como concatenamos nossos arquivos CSS. Também colocamos a chamada para o `stripDebug`, que já removeu nossos logs.

Agora, passaremos nossos arquivos por mais um processo de otimização, conhecido como `uglify`. Esse processo remove os espaços em branco do JavaScript que vamos gerar e os comentários,

comprime e, por fim, substitui o nome das variáveis por nomes menores (geralmente, letras ou números) para deixar nosso código o menor possível!

Vamos instalar o módulo que cuida disto para nós, o `npm install --save-dev gulp-uglify`, e importar em nosso `gulpfile.js` :

```
var gulp = require('gulp');
var concat = require('gulp-concat');
var minifyCSS = require('gulp-minify-css');
var rename = require('gulp-rename');
var autoprefixer = require('gulp-autoprefixer');
var stripDebug = require('gulp-strip-debug');
var uglify = require('gulp-uglify');
```

O módulo `uglify` pode receber um parâmetro de opções. Vamos detalhar um pouco sobre quais são e seus usos, pois podem ser bem importantes na hora de aplicá-lo em seu projeto pessoal:

1. `mangle` : recebe um booleano que vai controlar se as variáveis devem ou não ser renomeadas, a opção default é `true` . Tome cuidado ao usá-la, ela pode quebrar a compatibilidade com frameworks que dependem do nome da variável, como AngularJS em versão abaixo de 2.0 (ainda em beta durante a escrita deste livro).
2. `compress` : também recebe um booleano que deve controlar se os arquivos devem ou não ser comprimidos (remoção de espaço em branco e quebras de linha). Por padrão, assim como o `mangle` , essa opção vem ligada.
3. `preserveComments` : recebe uma `String` que indica se os comentários devem ou não ser mantidos no processo de `uglify` ; temos a opção `"all"` , que vai manter todos os comentários nos arquivos; e a opção `"some"` , que manterá todos os comentários que começarem com ponto de exclamação (!).

No nosso caso, queremos usar todas as opções e, por isso, não

precisamos definir nada ao colocar essa tarefa em nosso pipeline:

```
gulp.task('js', function() {
  var jsPath = {jsSrc:['./public/javascripts/**/*.js'], jsDest:'
public'};
  return gulp.src(jsPath.jsSrc)
    .pipe(concat('scripts.js'))
    .pipe(stripDebug())
    .pipe(uglify())
    .pipe(rename({ suffix: '.min' }))
    .pipe(gulp.dest(jsPath.jsDest));
});
```

Caso você quisesse, por exemplo, desativar o `mangle`, seu código ficaria assim:

```
gulp.task('js', function() {
  var jsPath = {jsSrc:['./public/javascripts/**/*.js'], jsDest:'
public'};
  return gulp.src(jsPath.jsSrc)
    .pipe(concat('scripts.js'))
    .pipe(stripDebug())
    .pipe(uglify({mangle: false}))
    .pipe(rename({ suffix: '.min' }))
    .pipe(gulp.dest(jsPath.jsDest));
});
```

Já adicionamos também o sufixo `.min` em nosso arquivo, que agora está pronto para ser importado em nossa view! Em nosso `layout.ejs`, vamos remover a chamada aos arquivos `.js` e adicionar somente o arquivo gerado pelo gulp.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %> </title>
    <link rel="stylesheet" type="text/css" href="styles.min.css">
    <!--[if lt IE 9]>
      <script src="http://html5shiv.googlecode.com/svn/trunk/html5.js"> </script>
    <![endif]-->
  </head>
  <body>
    <div class="container">
      <%- body %>
    </div>
```

```

    <script type="text/javascript" src="scripts.min.js"> </script>
  </body>
</html>

```

Caso você venha a enfrentar algum erro durante as tarefas no gulp, você vai reparar que as mensagens de erro nem sempre ajudam a resolver o problema. Para isto, existe o `guttil.log`, cujo uso é bem simples: basta adicionar uma chamada ao método `on` após adicionar algo ao `pipe`, este método recebe uma String e uma função. Em nosso caso, a função será o `guttil.log`, e a String será um evento que vamos observar, o evento de `error`.

O `guttil.log` nos fornece uma `stacktrace` melhor e mais detalhada, ficando mais fácil de encontrar e resolver os problemas que podem ocorrer ao processar nossos arquivos.

```

gulp.src(jsPath.jsSrc)
  .pipe(stripDebug().on('error', guttil.log))
  .pipe(concat('scripts.js').on('error', guttil.log))
  .pipe(uglify().on('error', guttil.log))
  .pipe(rename({ suffix: '.min' }))
  .pipe(gulp.dest(jsPath.jsDest));

```

Simples, não? Mas pode ser algo extremamente útil e poderoso para debugarmos nosso código.

Se formos ao nosso terminal e rodarmos `gulp js css`, ele executará as duas tarefas e produzirá os arquivos na pasta `public`. Vamos deixar mais fácil para executá-las, registrando uma tarefa padrão para o `gulp`. Isso se dá por meio de uma tarefa com o nome `default`; sempre que você criar uma tarefa com esse nome, poderá omitir seu nome no terminal.

```

gulp.task('default', ['css', 'js']);

```

Volte ao terminal e, dentro da pasta do projeto, digite apenas `gulp`. Veja que ele executou ambas as tarefas:

```

maia@maia:~/BA/my-todo: (master)$ gulp
[11:15:18] Using gulpfile ~/Documents/Projetos/my-todo/gulpfile.js
[11:15:18] Starting 'css'...
[11:15:18] Starting 'js'...
[11:15:21] Finished 'js' after 3.38 s
[11:15:21] Finished 'css' after 3.4 s
[11:15:21] Starting 'default'...
[11:15:21] Finished 'default' after 0.76 ms
maia@maia:~/BA/my-todo: (master)$

```

Figura 2.4: gulp working

Suba o servidor e verifique que está tudo OK.

Minha lista de tarefas

Nome:	Sobrenome:	Tarefa:	Gravar
Tarefa:			
Tarefa	Deletar		

Figura 2.5: Browser ok

Agora, toda vez que alterarmos um arquivo JavaScript ou CSS, precisaremos rodar o comando `gulp` para que ele processe os arquivos e, aí sim, veremos as alterações em nosso navegador.

Se este é um processo repetitivo, podemos automatizar através do nosso task runner. Podemos avisá-lo para ficar olhando as pastas onde nossos arquivos estão e, quando houver alguma alteração, ele rodará automaticamente as tarefas que cuidam dos arquivos que servimos ao navegador. Essa tarefa de olhar para as alterações é tão comum que está definida dentro do próprio gulp, não precisamos instalar ou importar nenhum módulo externo para isto.

```

gulp.task('default', ['css', 'js'], function() {
  // watch for JS changes
  gulp.watch('.public/javascripts/**/*.js', ['js']);
  // watch for CSS changes
  gulp.watch('.public/stylesheets/*.css', ['css']);
});

```

Rode o comando `gulp` novamente e veja que agora ele segura

nosso terminal. Altere um arquivo `.css` do projeto, como, por exemplo, o `form.css`, e verifique que o gulp vai realizar a tarefa `css` de forma automática para nós:

```
*Cecilio@ecowgMBA:~/my-todo$ gulp
[11:26:14] Using gulpfile ~/Documents/Projetos/my-todo/gulpfile.js
[11:26:14] Starting 'css'...
[11:26:14] Starting 'js'...
[11:26:17] Finished 'js' after 3.48 s
[11:26:17] Finished 'css' after 3.5 s
[11:26:17] Starting 'default'...
[11:26:17] Finished 'default' after 13 ms
[11:26:20] Starting 'css'...
[11:26:28] Finished 'css' after 379 ms
[11:26:24] Starting 'css'...
[11:26:24] Finished 'css' after 286 ms
```

Figura 2.6: gulps live reload

Podemos melhorar a performance do nosso gulp e ainda deixar nosso código mais enxuto por meio do `gulp-load-plugins`. Este módulo nos ajuda a gerenciar as importações, pois, a partir da versão 0.4.0, ele faz o lazy load dos outros módulos, ou seja, eles só são carregados no momento em que são chamados.

Para que ele possa gerenciar nossos módulos, eles precisam estar dentro da área de `development` do nosso `package.json`. Isso não será um problema para nós, pois sempre instalamos as dependências do nosso gulp utilizando a flag `--save-dev`.

A parte de `devDependencies` do seu `package.json` deve estar mais ou menos assim:

```
"devDependencies": {
  "gulp-autoprefixer": "^2.1.0",
  "gulp-concat": "^2.5.2",
  "gulp-minify-css": "^1.0.0",
  "gulp-rename": "^1.2.0",
  "gulp-strip-debug": "^1.0.2",
  "gulp-uglify": "^1.1.0"
}
```

Instale e importe o `gulp-load-plugins` do mesmo modo que

instalamos todos os outros módulos: `npm install --save-dev gulp-load-plugins`.

Mudaremos agora nosso `gulpfile.js` para carregar somente esse módulo, e deixar que ele gerencie a importação dos outros. O começo do seu arquivo deve estar assim:

```
var gulp = require('gulp');
var gulpLoadPlugins = require('gulp-load-plugins'),
    plugins = gulpLoadPlugins();
```

O `gulpLoadPlugins()` nos devolve um objeto JavaScript contendo todos os módulos que estiverem dentro de `devDependencies`. Para acessá-los, basta usar o padrão `camelCase`, como `plugins.minifyCss`

Nosso `gulpfile.js` ficará assim:

```
var gulp = require('gulp');
var gulpLoadPlugins = require('gulp-load-plugins'),
    plugins = gulpLoadPlugins();

gulp.task('css', function() {
  var cssPath = {cssSrc:['./public/stylesheets/*.css', '!*.min.css',
    '!/**/*.min.css'], cssDest:'public'};
  return gulp.src(cssPath.cssSrc)
    .pipe(plugins.concat('styles.css'))
    .pipe(plugins.autoprefixer('last 2 versions'))
    .pipe(plugins.minifyCss())
    .pipe(plugins.rename({suffix: '.min'}))
    .pipe(gulp.dest(cssPath.cssDest));
});

gulp.task('js', function() {
  var jsPath = {jsSrc:['./public/javascripts/**/*.js'], jsDest:'public'};
  return gulp.src(jsPath.jsSrc)
    .pipe(plugins.concat('scripts.js'))
    .pipe(plugins.stripDebug())
    .pipe(plugins uglify())
    .pipe(plugins.rename({ suffix: '.min' }))
    .pipe(gulp.dest(jsPath.jsDest));
});

gulp.task('default', ['css', 'js'], function() {
```

```
// watch for JS changes
gulp.watch('.public/javascripts/**/*.js', ['js']);
// watch for CSS changes
gulp.watch('.public/stylesheets/*.css', ['css']);
});
```

Veja que pouco mudou com a aplicação do nosso loader de plugins. Entretanto, agora podemos fazer o lazy load , além, é claro, de deixar nosso arquivo menor e mais elegante.

Avaliando nossas mudanças

Lembra-se de que inspecionamos no Google Chrome a quantidade de requisições feitas em nosso servidor ao carregarmos a página? Pois vamos realizar esse processo novamente, para verificar a eficiência de nossas alterações:

Name	Method	Status	Type	Initiator	Size	Time	Timing
scripts.min.js	GET	304	application/javascript	localhost:48	243 B	24 ms	
styles.min.css	GET	304	text/css	localhost:48	341 B	24 ms	
localhost	GET	304	text/html	localhost	149 B	8 ms	

Figura 2.7: Chrome Inspector

Podemos ver que agora temos apenas três requisições: uma para o HTML, uma para o JavaScript e uma última para o CSS. Reduzimos pela metade, neste caso, o número de requisições e, em termos de tamanho, reduzimos de 0.34mb para 0.24mb, uma melhora de 30%!

Pode parecer pouco, mas lembre-se de que a aplicação de exemplo é bem pequena: quanto maior a aplicação, maior o número de estilos e scripts e, por consequência, maior o ganho!

Melhorar o modo como servimos arquivos ao usuário é de extrema importância. Inclusive, uma das métricas do Google para seu sistema de ranking está justamente na velocidade com que os

sites carregam, principalmente se o usuário está acessando por meio de um celular.

2.2 REVISANDO

Neste capítulo, aprendemos sobre alguns problemas ao servir arquivos na web, vimos o que é o gulp e como ele nos ajuda a resolver esses problemas, reduzimos o número de requisições em nosso servidor e deixamos essas requisições mais leves por meio da concatenação e `uglify` dos arquivos. Ao final, ainda adicionamos um sistema de `watch` que facilita nosso processo de desenvolvimento, pois ele toma conta de aplicar as mudanças nos arquivos finais.

Se você quiser saber mais sobre outros plugins do gulp, você pode encontrar em <https://www.npmjs.com/search?q=gulp>.

O código do projeto e todo o código do capítulo pode ser visto em <https://github.com/CaioIncau/my-todo/tree/cap1>.

CUIDANDO DE ERROS E LOGS

3.1 TRABALHANDO COM OS ERROS HTTP

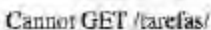
Agora que cuidamos de nossos arquivos estáticos, vamos nos preocupar com o gerenciamento de erros em nossa aplicação. No modo como ela está agora, caso algum erro aconteça, o usuário final verá uma tela nem um pouco agradável.

Precisamos começar a nos preocupar com isso. Inclusive, vemos um pedaço da nossa `stacktrace` na tela e, com isso, podemos expor algo que não deveríamos.

Vamos começar com o caso do erro 404. Este erro acontece quando o servidor não encontra a rota ou a página para ser renderizada, e é o erro mais comum e com maior chance de acontecer, pois, no geral, é causado pelo usuário.

Todos os erros HTTP que começam com 4xx, por definição, são causados pelo usuário.

Acesse uma URL que não existe e veja o que aparece:



Cannot GET /tarefas/

Figura 3.1: Página de erro

Uma página nada agradável e que não diz nada para o usuário comum que não entende de HTTP.

Para melhorar isto, vamos começar criando um `middleware`, que será responsável por cuidar exclusivamente dos erros da aplicação. Para isso, vamos criar um arquivo chamado `erros.js` na pasta `config`.

```
module.exports = function(app) {  
}
```

Não se esqueça de receber a `app` como parâmetro. Isso é importante, pois é por meio dela que responderemos aos erros.

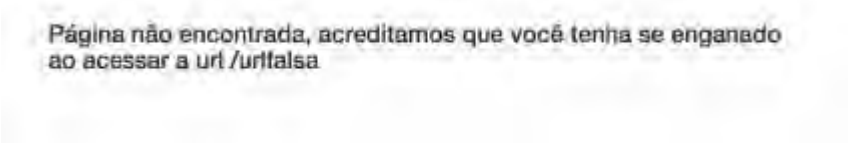
```
module.exports = function(app) {  
  app.use(function(req, res, next){  
    res.status(404).render('erro404', { url: req.url ,title : "P  
    ágina não encontrada"});  
  });  
}
```

Agora, nosso `middleware` recebe o `request` e o `response` (mais à frente, vamos abordar detalhes sobre estes dois objetos), e verifica se o `status` da resposta foi `404`; em caso positivo, ele renderiza uma view chamada `error404`.

Vamos criar uma view simples para isso:

```
<% layout( 'layout' ) -%>  
  
<h1 class="page-header">Página não encontrada, acreditamos que voc  
ê tenha se enganado ao acessar a url <%= url %></h1>
```

Suba o servidor e acesse uma URL que não existe. Verifique que você foi redirecionado para a página de erro.



Página não encontrada, acreditamos que você tenha se enganado ao acessar a url /urlfalsa

Figura 3.2: Nova página de erro

Muito melhor, não é mesmo? Agora, o usuário não recebe informações sobre HTTP que não dizem nada ao usuário final.

Assim, é hora de cuidar do erro mais temido pelos programadores, o erro 500.

Toda a linha de erros HTTP com código começando em 5XX é causado primariamente pelos programadores do serviço web. Por isso, vamos ser educados e pedir para o usuário nos contatar sobre este erro, para que assim possamos cuidar dele.

Você pode simular um erro em seu controller e causar um erro 500 propositalmente para testarmos o que vamos implementar: basta adicionar a seguinte linha em um método de nosso controller `index.js` :

```
next(new Error('erro simulado'));
```

Veja como fica o código completo do método `index` com a linha de erro adicionada. Não se esqueça de adicionar antes de o método tentar procurar no banco pelas tarefas.

```
exports.index = function ( req, res, next ){
  var user_id = req.cookies ?
    req.cookies.user_id : undefined;
  next(new Error('erro simulado'));
  Todo.
    find({ user_id : user_id }).
    sort( '-updated_at' ).
    exec( function ( err, todos ){
      if( err ) return next( err );


      res.render( 'index', {
        title : 'Minha lista de tarefas',
```

```

    todos : todos,
  });
});
};

```

Acesse a URL referente ao método que você alterou e veja o que acontece:



```

Error: Not Found
at exports.index (/Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/index.js:8:10)
at Layer.handle [as handle_request] (/Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/layer.js:71:2)
at next (/Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/route.js:100:13)
at Route.dispatch (/Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/route.js:11:3)
at Layer.handle [as handle_request] (/Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/layer.js:71:2)
at /Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/index.js:34:24
at Function.process_params (/Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/index.js:127:12)
at Function.handle (/Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/index.js:29:10)
at Function.handle (/Users/catinca/Documents/Projetos/my-todo/node_modules/express/lib/router/index.js:89:10)

```

Perceba que essa tela é ainda pior que a tela de 404. Nela, temos um pedaço da `stacktrace`, que, além de assustar o usuário comum, também pode expor parte de nossa aplicação para algum usuário mal intencionado.

O código para tratar o erro 500 é bem próximo do 404:

```

app.use(function(err, req, res, next) {
  res.status(500);
  res.render('erro500', {title : "Erro interno"});
});

```

Nossa página de erro:

```
<% layout('layout') -%>
```

```
<h1 class="page-header">Houve um erro ao processar sua requisição,
por favor entre em contato com o administrador do site</h1>
```

Veja que a única diferença para o 404 é que mudamos o código de erro verificado e a página que será renderizada.

Acesse a rota em que você adicionou esta linha e confira que nossa página foi renderizada:

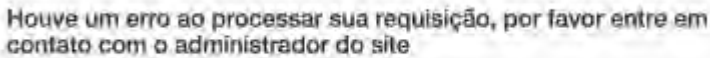


Figura 3.4: Nova página de erro

Poderíamos ter feito uma página mais detalhada, com a opção de contato ou uma imagem de fundo, mas só de escondermos a `stacktrace` do usuário, e dar um feedback melhor para ele, melhoramos muito.

O código completo de nosso `middleware`:

```
module.exports = function(app) {  
  app.use(function(req, res, next){  
    res.status(404).render('erro404', { url: req.url ,title : "P  
    ágina não encontrada"});  
  });  
  
  app.use(function(err, req, res, next) {  
    res.status(500);  
    res.render('erro500', {title : "Erro interno"});  
  });  
}
```

Poderíamos ainda tratar outros erros HTTP da forma que nos fosse conveniente. Veja alguns outros erros:

- **Erros do cliente 4xx:**

- 400: requisição inválida;
- 401: não autorizado (ocorre quando um usuário tenta acessar um recurso privado, limitado para administradores, ou ele não está logado);
- 402: pagamento necessário (esse erro quase nunca é usado);
- 403: proibido (ao contrário do 401, nesse caso o problema não é o nível de acesso do usuário, o recurso

acessado é simplesmente proibido);

- 404: o recurso não foi encontrado (o mais comum);
- 405: método HTTP não permitido (ocorre, por exemplo, quando tentamos acessar uma URL que só aceita `POST`).

- **Erros 5xx:**

- 500: erro interno no servidor (neste erro, a requisição foi válida, mas o servidor agiu de forma inesperada);
- 501: não implementada (o servidor não tem a função pedida);
- 502: Bad Gateway (ocorreu alguma problema de comunicação);
- 503: Service Unavailable (o serviço está indisponível no momento);
- 504: Time-out (ocorre quando uma requisição demora muito para ser resolvida);
- 505: versão HTTP não suportada (ocorre pouco hoje em dia, pois quase todas as aplicações já usam pelo menos HTTP 1.X).

OBSERVAÇÃO: não esqueça de remover a linha que joga o erro após o teste.

3.2 UM POUCO MAIS SOBRE O OBJETO REQUEST DO EXPRESS

O objeto `request` que recebemos na maioria de nossa aplicação (`res`) é nada mais do que um `wrapper` para o objeto do core do Node.js `http.request` . Este é uma representação das requisições

HTTP recebidas pelo servidor.

Ele possui basicamente quatro partes:

1. O tipo do método HTTP (GET , POST , PUT etc.)
2. **URI:** a URI acessada na requisição
3. **Header:** os parâmetros definidos no cabeçalho (header) da requisição.
4. **Body:** o conteúdo da requisição em si.

Vamos agora aos principais métodos desse objeto que, junto com o `Response` , é a base de nossa aplicação.

request.params

Com este método, podemos pegar os parâmetros passados para uma URL. Ele retorna uma lista com chave e valor (conhecida por alguns como mapa ou dicionário).

Veja um exemplo que usamos em nossa aplicação:

```
app.get('/destroy/:id', routes.destroy);
```

Nesse caso, podemos em nosso método `destroy` acessar o parâmetro `:id` por meio do `req.params.id`

request.query

Semelhante ao `params` , podemos pegar uma lista de chave/valor, mas, nesse caso, é uma lista que vem de uma Query String, como, por exemplo, `https://twitter.com/search?q=icaioincau&src=typd` .

Nesse caso, se tivéssemos acesso ao `req.query` , teríamos um objeto JavaScript assim:

```
console.log(request.query);  
//{q:'icaioincau', src:'typd'}
```

Geralmente, usamos Query String quando queremos apenas buscar uma informação no servidor, sem alterar.

request.body

Assim como os outros métodos, contém uma chave/ valor, mas com os dados enviados no corpo da requisição. Também já usamos em nossa aplicação, no método `create` do nosso `index.js` :

```
new Todo({
  user_id   : req.cookies.user_id,
  content   : req.body.content,
  name      : req.body.name,
  lastName  : req.body.lastName,
  updated_at : Date.now()
})
```

Construímos a tarefa a partir dos dados recebidos no corpo da requisição. Repare que as chaves são os `names` dos nossos `inputs` :

```
Nome:<input required="required" class="form-control" type="text"
name="name" />
Sobrenome:<input required="required" class="form-control" type="text"
name="lastName" />
Tarefa:<input required="required" class="form-control" type="text"
name="content" />
<input type="submit" value="Gravar" class="btn btn-default">
```

Por padrão, precisamos de um módulo que popule o `body` de nosso `request`. Em nossa aplicação, nós usamos o `body-parser`. Veja em nosso `app.js` que temos essas linhas:

```
app.use( bodyParser.json());
app.use( bodyParser.urlencoded({ extended : true }));
```

A primeira linha é usada para parsear dados da requisição quando recebemos um `json`.

Já a segunda define como popular o `body` a partir de requisições com o header `application/x-www-form-urlencoded`, ou seja, formulários.

request.cookies

Deste modo, podemos acessar os `cookies` de nosso navegador. Cookies são dados pequenos que ficam guardados em nosso navegador enquanto navegamos em um site, como o usuário logado ou os objetos em um carrinho de compras.

Esse método também trabalha com o formato chave/valor.

request.route

Por último, temos o método `route`, que nos provê acesso aos dados referentes a URI da requisição.

Podemos pegar a URL acessada (`req.route.path`), ou o método `http` (`req.route.method`).

O `route` é muito útil quando lidamos com `middlewares`.

request.ip

Com este método, podemos ver o `ip` do cliente que gerou a requisição, muito útil para ferramentas de log ou proteção.

request.get()

Por meio do método `get()`, podemos acessar os atributos do `header` de nossa requisição, e podemos, por exemplo, acessar o `User Agent` do usuário: `app.get("User-Agent")`

Também temos o método `request.header()`, que tem a mesma função, mas este não diferencia maiúsculas de minúsculas, ao contrário do `.get()`

3.3 UM POUCO MAIS SOBRE O OBJETO

RESPONSE DO EXPRESS

Assim como o objeto `request` do Express, o `response` é um wrapper do `http.response` do Node.js, mas com alguns adicionais.

Podemos ver esse objeto como uma extensão do `response` original do node.

Assim como estudamos o Request, veremos os principais métodos do `response`.

response.render

O método `.render` pode receber três parâmetros, sendo que apenas o primeiro é obrigatório, sendo eles: `name`, `data` e `callback`.

O `name` é o nome do seu arquivo que será renderizado. Este não precisa ter a extensão declarada e deve estar na pasta que você configurou no `app.js`:

```
app.set( 'views', path.join( __dirname, 'views' ) );  
app.set( 'view engine', 'ejs' );
```

Em nosso caso, na pasta `views`, exemplo de uso do `.render` com um parâmetro:

```
res.render( 'index' );
```

Deste modo, ele vai buscar na pasta `views` pelo arquivo `index.ejs`

Em nossa aplicação, nós usamos também o parâmetro `data`, que passa para a nossa view os dados que serão renderizados, tornando-a mais dinâmica:

```
res.render( 'index', {  
  title : 'Minha lista de tarefas'  
});
```

O parâmetro `data` é um chave-valor comum, e cada tipo de template engine lida com isto de um modo diferente na view.

Temos ainda o terceiro parâmetro, que não usamos na aplicação de exemplo. Este é uma função de callback que recebe um objeto que representa os possíveis erros e outro que é o próprio HTML que será gerado.

```
app.get('/', function(req, res) {  
  res.render('index', {title: 'Minha lista de Tarefas'}, function  
(error, html) {  
    //fazemos algo com o erro ou o html  
  }); });
```

Aqui seria possível, por exemplo, imprimir o erro.

response.locals

O `response.locals` serve para passar dados para a view, assim como o `data` do `render`.

```
res.locals = { title: 'Minha lista de tarefas' };  
res.render('index');
```

Este código é equivalente a este:

```
res.render( 'index', {  
  title : 'Minha lista de tarefas'  
});
```

response.set

Assim como no `request.set()`, este método serve para adicionarmos valores no `header` de nossa resposta.

Por exemplo, podemos avisar ao navegador que nossa resposta será em formato `html`:

```
res.set('Content-Type', 'text/html');
```

response.send

O `response.send` recebe uma `String` ou um objeto `JSON`, e responde como este valor. Esse método é bastante útil para APIs, por exemplo:

```
res.send( {  
  title : 'Minha lista de tarefas',  
  todos : todos,  
});
```

Isso nos mostrará a seguinte tela:

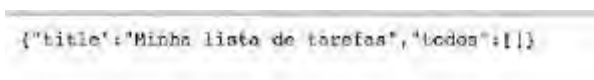


Figura 3.5: Minha lista de tarefas

Desse modo, poderíamos ter uma outra aplicação que pegaria a lista de tarefas e faria alguma outra coisa, como enviar um e-mail com a lista.

response.redirect

Este método recebe uma `String` que é uma URL, e redireciona o usuário para essa URL, por exemplo:

```
res.redirect("/");
```

Este código envia o usuário para a `home`.

Podemos também usar uma URL externa:

```
res.redirect("http://www.twitter.com");
```

Este código envia o usuário para a página inicial do Twitter

Agora, você já viu as principais funções do `request` e `response`, e isto deverá ser o suficiente para a maioria do seu uso no dia a dia como programador. Porém, existem ainda outros métodos menos

utilizados, você pode consultá-los em <http://expressjs.com/api.html>.

3.4 MANTENDO HISTÓRICO DE NOSSA APLICAÇÃO

Queremos agora manter um histórico de acessos em nossa aplicação, pois isso é muito importante para identificar atividades suspeitas, guardar quais URLs deram problema e até mesmo controlar que tipo de pessoa está acessando determinado recurso de nosso site.

O modo mais simples seria adicionar um `console.log` em todos os nossos métodos, e eles imprimirem quando foram acessados. Entretanto, não nos parece efetivo usar logs desse modo, certo? Acredite que ainda existem pessoas fazendo coisas deste tipo.

Outro modo que seria bem mais efetivo seria a criação de um `middleware` que cuidasse disso para a gente. Este, na verdade, seria um ótimo modo, mas assim como toda tarefa corriqueira do `express`, já temos alguém que cuida disso, e seu nome é Morgan.

Usando Morgan

O `morgan` é um módulo que cuida de gerar logs das `urls` que apresentaram erro, algo simples, mas que é bastante útil.

Vamos começar adicionando-o ao projeto por meio do nosso `npm`. Vamos rodar o comando `npm install morgan` em nosso terminal.

A principal vantagem de usarmos o `morgan` é que ele nos fornece vários modos de saída para nosso log de requisições.

Vamos importá-lo em nosso `app.js`:

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

Agora, registre-o na aplicação:

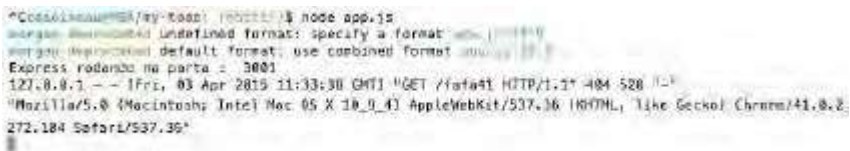
```
//resto do app.js
app.use(morgan());
require( './config/errors' )(app);

http.createServer( app ).listen( app.get( 'port' ), function (){
  console.log( 'Express rodando na porta : ' + app.get( 'port' ) )
;
});
```

Repare que colocamos ele logo acima da configuração de erros e do nosso `createServer`. É sempre bom lembrar de que a ordem importa em nosso `app.js`. Os `middlewares` serão resolvidos de acordo com a ordem em que foram especificados, por isso deixamos o `morgan` no final, onde todas as rotas já foram resolvidas. Se você colocar no começo do arquivo, por exemplo, ele não saberá as rotas definidas e não vai funcionar.

O único que ficou abaixo dele foi o nosso controlador de erros (`./config/errors`), pois queremos que o `morgan` guarde todas as requisições que falharem antes de tratarmos erros.

Acesse uma URL inexistente – ou seja, que gere um erro 404 –, e veja que o `morgan` gerou um log em seu terminal.

A screenshot of a terminal window with a dark background. The first line shows a command prompt and the command 'node app.js'. The second line shows a Morgan log entry: 'morgan: undefined default format; use combined format' followed by a timestamp and IP. The third line shows an Express log entry: 'Express rodando na porta : 3001'. The fourth line shows an HTTP log entry: '127.0.0.1 - - [Fri, 03 Apr 2015 11:33:38 GMT] "GET /fafa41 HTTP/1.1" 404 528 "-"'. The fifth line shows a user agent string: '"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.184 Safari/537.36"'.

```
^C$ cd /Users/rafael/Projects/express-test$ node app.js
morgan: undefined default format; use combined format
Express rodando na porta : 3001
127.0.0.1 - - [Fri, 03 Apr 2015 11:33:38 GMT] "GET /fafa41 HTTP/1.1" 404 528 "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.184 Safari/537.36"
```

Figura 3.6: Log do morgan

Vamos analisar bem nosso log: ele nos mostra o IP que acessou nosso servidor – neste caso, `127.0.0.1` –, a data `[Fri, 03 Apr 2015 11:33:38 GMT]`, seguido pelo método `HTTP` e a URL `"GET /fafa41 HTTP/1.1"`. Depois temos a resposta `HTTP 404`, o tamanho da resposta `528` e, por fim, a plataforma do cliente: seu Sistema

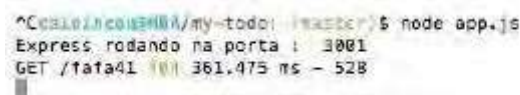
Operacional e seu navegador – Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.104 Safari/537.36

Geramos um log bem completo, certo? Mas no desenvolvimento, em que ocorrem muitos erros, isso pode acabar poluindo nosso terminal e nos atrapalhando.

Para evitar esse tipo de problema, podemos passar o parâmetro "dev" para nosso morgan, fazendo ele gerar logs menores para nós. Altere seu `app.js`, e mude o `morgan` para logar deste modo.

```
app.use(morgan("dev"));
```

Acesse novamente uma URL que retorne 404, e veja a saída.

A terminal window showing a command prompt where the user has run 'node app.js'. The output shows 'Express rodando na porta : 3001' followed by a log entry: 'GET /fafa41 404 361.475 ms - 528'.

```
^C$ node app.js
Express rodando na porta : 3001
GET /fafa41 404 361.475 ms - 528
```

Figura 3.7: Novo log do morgan

Repare que agora temos menos informações: só nos é mostrado o método HTTP, seguido da URL, a resposta, o tempo de resposta e o tamanho dela.

Bem mais limpo e fácil de ler, e algo útil para o ambiente de desenvolvimento, onde não precisamos de tantas informações do usuário que, no caso, é o programador.

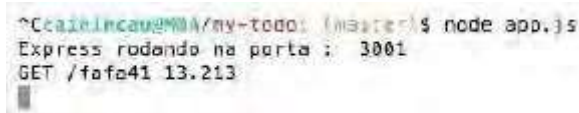
Além desses dois padrões, temos alguns outros, como:

- `combined`, que é o mesmo padrão que os servidores Apache usam;
- `common`, que é o equivalente a deixarmos em branco;
- `short`, que tem menos informação que o `common`, mas loga o `response-time` ; e
- `tiny`, que oferece o mínimo possível de informações.

Ter várias opções de padrão é algo muito interessante, mas o mais interessante mesmo é que podemos customizar nossa ferramenta de logs com um padrão que achamos melhor.

Vamos analisar um exemplo onde só queremos saber qual o método HTTP foi usado, qual URL foi acessada e o tempo de resposta:

```
app.use(morgan(':method :url :response-time'))
```

A terminal window showing the output of a Node.js application. The first line is a command prompt: ^Ccairn@cairn:~/my-todo: (master) \$ node app.js. The second line is the message 'Express rodando na porta : 3001'. The third line is a log entry: 'GET /fafa41 13.213'.

```
^Ccairn@cairn:~/my-todo: (master) $ node app.js
Express rodando na porta : 3001
GET /fafa41 13.213
```

Figura 3.8: Exemplo de log

Veja que o log ficou bem pequeno. O `morgan` pode receber várias chaves que indicam o que deve ser logado e qual sua sequência, as opções disponíveis são:

1. `:date[format]` : loga a data, os formatos aceitos são `iso` , `clf` e `web` .
2. `:http-version` : indica a versão HTTP usada na requisição.
3. `:method` : indica qual método HTTP foi usado na requisição.
4. `:referrer` : indica a URL de origem para a qual redirecionou a URL de erro; essa informação é muito útil para encontrarmos onde estão linkando para nossas páginas de 404.
5. `:remote-addr` : este dado é o IP do usuário.
6. `:req[header]` : o atributo `header` da requisição.
7. `:res[header]` : o atributo `header` da resposta.
8. `:response-time` : o tempo de resposta da URL, este tempo é impresso em milissegundos.
9. `:status` : o código da resposta.
10. `:url` : a URL acessada que originou algum erro.
11. `:user-agent` : os detalhes do usuário, como sistema operacional e navegador, bem útil para encontrarmos problemas que

acontecem em plataformas específicas.

Você pode combinar quantas opções quiser e na ordem que desejar.

Agora que sabemos como guardar as URLs que deram problema, ficará mais fácil de detectarmos e resolvermo-nos. Mas espere, onde ficam esses erros? No nosso terminal, certo? Entretanto, não ficamos com o terminal aberto ao colocarmos nossa aplicação em produção, como vamos resolver isto?

Bem, para nossa sorte, o `morgan` nos permite gravar seu log em um arquivo, assim mantemos histórico.

Para isto, vamos usar também o módulo `FileSystem` ou `fs`. Não abordaremos esse módulo em detalhes por ser fora do escopo deste livro, vamos apenas explicar a função usada neste caso.

Vamos alterar nosso `app.js` para importar o `fs`.

```
//resto dos imports  
var fs = require('fs')
```

Repare que não precisamos instalar o `fs` pelo NPM, pois este é um `core module`, ou seja, já vem embutido na sua instalação do Node.

Vamos agora criar um arquivo de logs e pedir para o `morgan` colocar lá todo seu `output` :

```
var accessLogStream = fs.createWriteStream(__dirname + '/urls.log'  
, {flags: 'a'})  
app.use(morgan(':method :url :response-time', {stream: accessLogStream}));  
  
require( './config/errors' )(app);  
http.createServer( app ).listen( app.get( 'port' ), function () {  
  console.log( 'Express rodando na porta : ' + app.get( 'port' ) )  
;  
});
```


Vamos analisar com calma esse código. Primeiro, nós chamamos o método `fs.createWriteStream`, que recebe o caminho para um arquivo – neste caso, nosso arquivo de log – e um objeto com a opção `flags:a`. Esta última opção serve para indicar que o arquivo será escrito em `append mode`. Isso significa que toda vez que alguém usar este arquivo, deve adicionar conteúdo ao final dele, e não sobrescrever.

Depois, passamos para o `morgan` a opção `stream`. Veja que criamos um `WriteStream`, que nada mais é do que um objeto capaz de escrever em um arquivo. Feito isso, podemos subir nosso servidor e acessar algumas URLs inexistentes.

Verifique que foi criado um arquivo `urls.log` na raiz de seu projeto. Abra-o com o seu editor de texto favorito, e verifique que o log está lá.



```
1 GET /fafa41 12.508
2 GET /ops 52.174
3 GET /nope 1.622
4 GET /why? 1.878
5
```

Figura 3.9: Log no seu editor de texto

Agora sim, guardamos de forma satisfatória todas as URLs que geraram algum tipo de erro em nossa aplicação. Manter históricos de problemas é fundamental para qualquer aplicação.

3.5 AMBIENTES DE DESENVOLVIMENTO

Se você já desenvolveu em outra linguagem, provavelmente já sabe o conceito de `environment`, que, em tradução direta, significa "ambiente". Mas o que seria um ambiente em desenvolvimento de software?

O ambiente de desenvolvimento de um software nada mais é do que o termo para englobar todos os itens de que o projeto precisa para desenvolver e implementar o sistema, como, por exemplo, ferramentas, variáveis, templates e infraestrutura.

Por exemplo, em nosso ambiente de desenvolvimento, queríamos que o log fosse pequeno para evitar sujar nossa saída no terminal, mas em produção gostaríamos de um log mais robusto, com mais detalhes e informações sobre o usuário.

Em vez de lembrarmos de alterar a linha que faz o log toda a vez que formos enviar nossa aplicação para produção, nós podemos ter variáveis que cuidem disso para nós.

Para mudarmos de ambiente em Windows, usamos o comando `set NODE_ENV=seuEnvironment`. Os ambientes mais comuns são `test`, `development` e `production`.

Para mudarmos de ambiente em sistemas Unix, usamos o comando `export NODE_ENV=seuEnvironment`. Para ter acesso ao `environment` em nossa aplicação Node, usamos o método `app.get('env')`;

Vamos agora fazer com que o `morgan` trabalhe de forma diferente, dependendo do seu ambiente de desenvolvimento:

```
var env = app.get('env');
```

Deste modo, conseguimos descobrir o ambiente em que estamos trabalhando.

```
var env = app.get('env');
```

```
if(env=="production"){
```

```
}else{
```

```
}
```

Podemos comparar com uma `String` , normalmente, aí decidiremos o que fazer em cada ambiente.

```
var env = app.get('env');

app.use(static(path.join( __dirname, 'public')));

if(env=="production"){
    var accessLogStream = fs.createWriteStream(__dirname + '/urls.
log', {flags: 'a'})
    app.use(morgan('combined', {stream: accessLogStream}));
}

if(env=="development"){
    app.use(morgan());
}
```

Em caso de produção, vamos usar o método `combined` , que, como explicado anteriormente, é o padrão de servidores Apache. Além disso, vamos escrever em arquivo.

Em caso de desenvolvimento, vamos usar o padrão do `morgan` .

Sempre é interessante usar um `else` após checar todos os ambientes que terão regras diferenciadas, pois podemos ter novos ambientes no futuro que não ficariam cobertos por códigos sem o caso específico. Vamos refatorar o código e colocar um caso default.

```
if(env=="production"){
    var accessLogStream = fs.createWriteStream(__dirname + '/urls.
log', {flags: 'a'})
    app.use(morgan('combined', {stream: accessLogStream}));
}else{
    app.use(morgan());
}
```

Pronto, agora, caso apareçam novos ambientes, eles usarão o `morgan` em seu estado default também.

Suba o servidor em ambos os modos, e verifique que o formato de log mudou.

Você pode alterar o ambiente e subir o servidor na mesma linha

com `export NODE_ENV=development && node app` ou `set NODE_ENV=development && node app`, dependendo do seu sistema operacional.

Você pode usar este controle de ambiente para várias coisas, como, por exemplo, enviar ou imprimir o e-mail no terminal, expor ou não uma rota de acordo com o ambiente etc.

```
if(env=="development"){  
  app.get( '/limpaOBancoDeDados', routes.limpaOBanco );  
}
```

As possibilidades são quase infinitas, só depende de sua necessidade.

3.6 REDUZINDO A QUANTIDADE DE ERROS COM EXPRESS-VALIDATOR

Boa parte dos erros é ocasionada por dados inseridos de forma errada em sua aplicação, seja por formulário ou query string. Esses dados são processados por nossa aplicação e, durante esse processamento, acaba por acontecer algum erro.

Seria interessante validarmos esses dados antes de serem inseridos, certo?

Acesse nossa aplicação e tente salvar uma tarefa sem preencher os dados.

Minha lista de tarefas

Nome:	Sobrenome:	Tarefa:	Gravar
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="button" value="Gravar"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="button" value="Gravar"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="button" value="Gravar"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="button" value="Gravar"/>

Figura 3.10: Salvando uma tarefa sem os dados

Veja que é possível! Não cuidamos disto. Se algum método usar o nome da tarefa, como um `split`, receberíamos um erro, pois não estará definido.

É boa prática validarmos os dados tanto no lado do servidor quanto do cliente. Para isso, vamos adicionar o atributo `required="required"` em nosso formulário, tanto no `edit.ejs` quanto no `index.ejs`

Nosso `edit.ejs` :

```
<% layout( 'layout' ) -%>

<h1 class="page-header"><%= title %> </h1>

<% todos.forEach( function ( todo ){ %>
  <form action="/update/<%= todo._id %>" method="post" class="
form-create navbar-form navbar-left" accept-charset="utf-8">
    <div class="form-group">
      Nome:<input required="required" class="form-control" type="
text" name="name" value="<%= todo.name %>" />
      Sobrenome:<input required="required" class="form-control" t
ype="text" name="lastName" value="<%= todo.lastName %>" />
      Tarefa:<input required="required" class="form-control" type=
"text" name="content" value="<%= todo.content %>" />
    </div>
    <input type="submit" value="Gravar" class="btn btn-default">
    <a href="/destroy/<%= todo._id %>" title="Delete esse item" c
lass="btn btn-default">Delete </a>
  </form>
```

```
<% }); %>
```

Nosso index.ejs :

```
<% layout( 'layout' ) -%>
```

```
<h1 class="page-header"><%= title %></h1>
```

```
<div id="list">
```

```
  <form action="/create" method="post" accept-charset="utf-8" class="navbar-form form-create">
```

```
    <div class="form-group">
```

```
      Nome:<input required="required" class="form-control" type="text" name="name" />
```

```
      Sobrenome:<input required="required" class="form-control" type="text" name="lastName" />
```

```
      Tarefa:<input required="required" class="form-control" type="text" name="content" />
```

```
        <input type="submit" value="Gravar" class="btn btn-default">
```

```
    </div>
```

```
  </form>
```

```
</div>
```

```
<div class="panel panel-default">
```

```
  <!-- Default panel contents -->
```

```
  <div class="panel-heading">Tarefas</div>
```

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

```
  <table class="table">
```

```
    <tr>
```

```
      <th>Tarefa</th>
```

```
      <th>Deletar</th>
```

```
    </tr>
```

```
    <% todos.forEach( function ( todo ) { %>
```

```
      <tr>
```

```
        <td><a href="/edit/<%= todo._id %>" title="Atualize este item"><%= todo.content %></a></td>
```

```
        <td><a href="/destroy/<%= todo._id %>" title="Delete este item">Delete</a></td>
```

```
      </tr>
```

```
    <% }); %>
```

```
  </table>
```

```
</div>
```

Repare que ambos continuam idênticos, só adicionamos o required . Tente agora salvar uma tarefa sem preencher seus

campos:



Figura 3.11: Agora com required

Agora o navegador não deixa enviar o formulário com nome vazio. Ainda sim, poderíamos sofrer com um post feito de outro modo; ou, pior ainda, navegadores antigos não suportam o atributo `required`.

Mas este não é o único caso que queremos validar, logo, vamos fazer isso do lado do servidor e aprender como validar os diferentes casos.

Para realizar esse trabalho, temos o `express-validator`. Instale-o no seu projeto `npm install express-validator`.

Agora, importe e registre o validador no seu `app.js`. Lembre-se de que ele precisa ser registrado após o `bodyParser`:

```
//imports
var validator = require('express-validator');
//imports

//resto do app.js
app.use( bodyParser.json() );
app.use( bodyParser.urlencoded({ extended : true }) );
app.use(validator());
//resto do app.js
```

Agora com o `validator` registrado em nossa aplicação, podemos começar a usar a validação do lado do servidor.

Ao adicionarmos este módulo no projeto, nosso objeto `request` ganha três novos métodos. O `checkBody`, que procura por um valor

no `body` da requisição e adiciona uma mensagem de erro, caso a validação não funcione. Similar a este, temos o `checkParams` e o `checkQuery`, ambos têm a mesma funcionalidade do `checkBody`, mas um procura no `req.params` e o outro no `req.query` (parâmetros recebidos via `GET`).

Veja um exemplo:

```
req.checkBody('nome', 'Nome inválido');
```

Deste modo, definimos onde o valor deve ser procurado, qual o valor a ser procurado – nesse caso, `nome` –, e qual a mensagem de erro deve ser chamada se a verificação falhar – neste caso, `Nome inválido`.

Mas espera aí, ainda não definimos o tipo de verificação a ser feita!

Isto é bem simples. É só encadear uma ou mais chamadas ao final do nosso `check`. Veja um exemplo:

```
req.checkBody('email', 'Email inválido').isEmail();
```

Podemos realizar mais de uma verificação por vez, basta encadearmos mais um método:

```
req.checkBody('email', 'Email inválido, pode ser opcional').optional().isEmail();
```

Vejamos alguns métodos interessantes de validação:

- `.contains(String)` : verifica se o valor contém uma `String` que passamos como parâmetro.
- `.isURL` : verifica se o valor é uma URL.
- `.isAlpha` : verifica se o valor possui apenas `Strings` de A-Z.
- `.isNumeric` : verifica se o valor contém apenas número.
- `.isAlphanumeric` : equivalente ao 3 e 4 juntos.

- `isHexColor` : verifica se o valor é uma cor hexadecimal.
- `isLowercase` : verifica se o valor é uma `String` com letras minúsculas.
- `isUppercase` : verifica se o valor é uma `String` com letras maiúsculas.
- `isInt` : verifica se o valor é um número inteiro.
- `isFloat` : verifica se o valor é um número de ponto flutuante.
- `isNull` : verifica se o valor é nulo
- `.len(Int, Int)` : verifica se o valor tem um tamanho mínimo e máximo.
- `.isMongoId` : verifica se o valor é um `Mongo ObjectId` (bem interessante para quem usa MEAN Stack).
- `.isDate` : verifica se o valor é uma data.
- `.isIn(arrayDeStrings)` : verifica se o valor está presente em um array de `Strings` passadas como parâmetro.

Temos todos esses validadores, mas caso eles não sejam suficientes para você, podemos criar nossos validadores customizados ao registrarmos o `expressValidator` em nossa aplicação. Veja como fica o registro de nosso módulo na aplicação:

```
app.use(validator({
  customValidators: {

  }
}
}));
```

E agora, como definimos estes validadores? É bem simples:

```
app.use(validator({
  customValidators: {
    isArray: function(value) {
      return Array.isArray(value);
    },
    gte: function(param, num) {
      return param >= num;
    },
  },
}));
```

```

    isCaio:function(value){
      value == "Caio";
    }
  }
});

```

Veja como é simples registrar validadores customizados. Usamos o mais famoso padrão do JavaScript, a chave-valor, na qual a chave será o nome do método de validação e o valor uma função que deve sempre retornar um valor booleano.

Agora que já aprendemos como usar o `express-validator` e até mesmo como criar validações customizadas, vamos implementá-lo em nosso projeto de exemplo.

Vamos em nosso `index.js` para garantir que o nome não seja vazio:

```

exports.create = function ( req, res, next ){
  req.checkBody('name', 'Nome é necessário').notEmpty();
  //resto do método

```

Até aqui nada novo, fizemos a verificação assim como já vimos.

```

exports.create = function ( req, res, next ){
  req.checkBody('name', 'Nome é necessário').notEmpty();
  errors = req.validationErrors();
  //resto do método.

```

Agora temos algo novo! O método `.validationErrors()`, que nos devolve uma lista de erros. Cada erro é um objeto JavaScript que contém três atributos:

- `param`, que é o nome do passado para verificação, em nosso caso sendo `name` ;
- `msg`, que é a mensagem de erro, em nosso caso sendo `Nome é necessário` ;
- `value`, que representa o valor passado para verificação, o que veio em nosso `request`, em nosso caso, o valor que preencheremos no formulário.

Precisamos verificar se houve algum erro, ou seja, se a lista está vazia ou não. A partir daí, redirecionamos o usuário para uma página de erro ou para a mesma página, mas mostrando os erros.

Em nosso caso, vamos redirecionar para uma nova página:

```
exports.create = function ( req, res, next ){
  req.checkBody('name', 'Nome é necessário').notEmpty();
  errors = req.validationErrors();

  if (errors) {
    res.status(400).render('erro400', { errors: errors ,title :
"Página não encontrada"});
    return;
  }
  //resto do método
```

Vamos usar uma página de Bad Request (erro 400), um erro que ocorre quando a requisição não é feita como o esperado, neste caso, faltando valores. Veja como fica nossa página de erro:

```
<% layout( 'layout' ) -%>

<div class="container" >
<% errors.forEach( function ( error ){ %>
  <p class="alert alert-danger" role="alert"> <%= error.msg %>
  <% }); %>

</div>
```

Usamos o mesmo layout das outras e percorremos a lista de erros, algo bem simples.

Suba sua aplicação e envie o formulário sem nome. Verifique que será redirecionado para a página de erro:



Figura 3.12: Página de erro

Se voltarmos ao começo, vemos que a tarefa não foi salva, exatamente o comportamento que queríamos.

Pronto, agora temos validação tanto do lado do servidor quanto do lado do cliente. Além de melhorarmos a experiência do usuário, ainda temos uma proteção extra para nossa lógica que não receberá dados inválidos.

3.7 REVISANDO

Neste capítulo, nós aprendemos a lidar com os principais erros HTTP, caso eles ocorram, e a como oferecer uma melhor experiência para o usuário.

Vimos também um pouco mais sobre os objetos `request` e `response` do Express.js, que, apesar de simples e muito usados em nosso dia a dia, na maioria das vezes de forma instintiva, valem o reforço, pois são as bases do Express.

Também aprendemos a manter logs de nossa aplicação para identificarmos problemas, trabalhando com diferentes ambientes de desenvolvimento (environments), e customizamos nossa ferramenta de log.

Por fim, validamos os dados que nossa aplicação recebe para reduzir o número de erros possíveis, feito tanto do lado do servidor quanto do cliente. É sempre prudente programar de modo defensivo.

Agora, temos uma aplicação com menor chance de erros e, caso eles ocorram, o usuário vai sofrer menos em sua experiência, e nós conseguiremos detectar e resolver os problemas com maior facilidade.

MELHORANDO PERFORMANCE E SEGURANÇA

Agora que já cuidamos dos nossos erros e logs, a aplicação está bem mais consistente. Ainda assim, podemos encontrar problemas de performance e segurança.

Vamos evoluir nossa aplicação com algumas técnicas que podem nos ajudar com esses problemas muito comuns em web apps.

4.1 GZIP

Não seria interessante deixar nossos *assets* ainda menores? Mas isso parece meio complicado, não? Já aplicamos tantas técnicas para diminuir seu tamanho.

Entretanto, ainda existe uma técnica no lado do servidor, a de comprimir o arquivo ao servi-lo: o navegador recebe o arquivo comprimido, descomprime e mostra. Você verá que, apesar de ser uma técnica extremamente poderosa, ela também é muito simples de ser aplicada em nossos servidores Node.

Existem inúmeros padrões de compressão de arquivos, como RAR , ZIP , 7ZIP e GZIP . Este último é o padrão que os navegadores

utilizam, e já vamos entender o porquê.

GZIP é a abreviação de GNU Zip, um compactador de dados baseado no algoritmo DEFLATE. O GZIP pode ser aplicado em qualquer fluxo de bytes, mas funciona melhor em recursos de texto como JavaScript, CSS e HTML. Neste tipo de arquivo, é possível conseguir uma taxa de compressão de 70% - 90%! Infelizmente, para imagens e afins, sua compressão é quase nula.

Todos os navegadores modernos têm suporte ao GZIP, inclusive o IE6.

Para usarmos essa compressão, vamos precisar do módulo `compression`, que vamos instalar por meio do comando: `npm install compression --save`.

Agora, assim como os outros módulos que usamos, vamos importá-lo em nosso `app.js`:

```
//resto dos imports
var compression = require('compression');
```

Para ligar a compressão, basta adicionar o módulo em sua app:

```
app.use(compression());
```

Suba o servidor e verifique se funcionou. Para isto, abra o Google Chrome e inspecione a página com o Developer Tools (botão `F12` em Windows/Linux ou `Cmd+Option+I` em Mac); vá à aba `Network` e clique em algum arquivo CSS, HTML ou JS. Veja que agora temos um `response header` e um `Content-Encoding: gzip`, isso nos mostra que o arquivo foi servido corretamente.

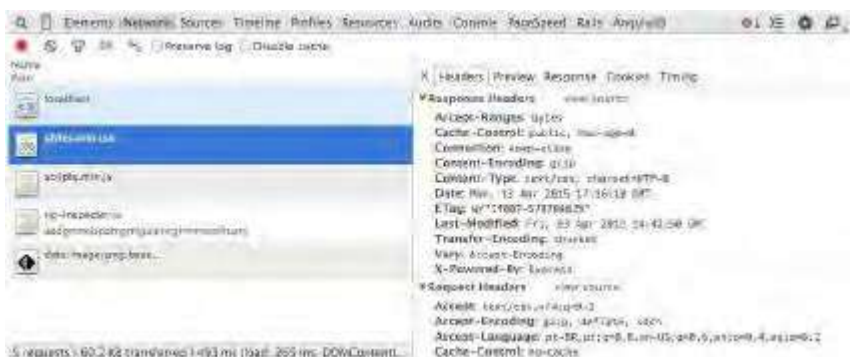


Figura 4.1: Response header

Podemos, ainda na aba **Network**, verificar a efetividade de nossa compressão.

Name	Path	Method	Status	Type	Initiator	Size Compressed	Time	Latency	Timeline
localhost		GET	200 OK	text/html	Other	1.4 KB	12 ms		
styles.min.css		GET	200 OK	text/css	localhost:9	20.7 KB	11 ms		
scripts.min.js		GET	200 OK	applicatio...	localhost:93	38.1 KB	32 ms		

Figura 4.2: Na aba network

Veja na coluna **Size**. Lá temos um número mais escuro, que é o tamanho do arquivo servido, e um número mais claro, que é o tamanho real do arquivo sem compressão.

Nosso HTML diminuiu de 1.4KB para 1.2KB; pouco ganho, pois o arquivo já era muito pequeno.

Agora, repare em nosso arquivo de CSS, ele diminuiu de 124KB para 20KB; uma redução de 620%! Já nosso JavaScript sofreu uma redução de quase 300%.

Lembre-se de que nossa aplicação começou com 340KB, antes de usarmos o Gulp ou GZIP, agora ela está em 60KB. Podemos

servir quase seis vezes mais clientes com a mesma largura de banda de nosso servidor. Além disto, entregamos o site seis vezes mais rápido para o cliente, que, por sua vez, fica mais contente com o tempo de resposta, além de consumir menos do plano de dados de conexões 3G.

Visto como usar o `gzip`, agora vamos aprender algumas customizações do módulo de compressão.

A primeira opção é o `filter`, no qual podemos decidir regras para quais arquivos serão comprimidos.

```
app.use(compression({filter: deveComprimir}))

function deveComprimir(req, res) {
  if (req.headers['x-no-compression']) {
    return false
  }

  // aqui temos o caso default, sempre importante tratarmos o caso
  default.
  return compression.filter(req, res)
}
```

Vamos agora entender como funciona o código. Por meio da opção `filter`, passamos uma função que decide qual tipo de arquivo deve ser comprimido.

A seguir, definiremos a função que recebe o `request` e o `response`, e fazemos um `if` para verificar a presença de um `Header`. Caso este esteja presente, não vamos comprimir.

```
if (req.headers['x-no-compression']) {
  return false
}
```

Logo a seguir deixamos o caso padrão. Sempre precisamos nos lembrar de cuidar do caso padrão, que não cai no nosso `if`.

```
return compression.filter(req, res)
```


Neste caso, deixamos o filtro que já vem embutido no módulo decidir o que deve ou não ser comprimido.

A segunda opção de customização é o `level`, no qual podemos alterar o nível de compressão que desejamos. Este número varia de -1 (que é um atalho para o valor padrão), passando por 0 (que é sem compressão) e chegando até 9 (que seria a maior taxa de compressão).

```
app.use(compression({level: 9}));
```

O valor padrão é 6. Quanto maior o número, maior será a compressão, porém, ela demorará mais tempo para ser executada, por isso costumamos deixar no meio termo. Ainda assim, dependendo da sua necessidade, você pode alterá-lo.

A última opção que veremos é o `memLevel`, que define quanta memória deve ser alocada para compressão. Seus valores variam de 1 até 9, tendo como padrão o valor 8.

```
app.use(compression({level: 9,memLevel: 9}));
```

Esta é uma opção bem avançada e deve ser usada com parcimônia.

4.2 USANDO CACHE DE NAVEGADOR

Uma ferramenta importante implementada em todos os navegadores modernos é a capacidade de guardar arquivos em cache.

Deste modo, o navegador não precisa carregar o mesmo arquivo a cada requisição, bem útil para reduzir o tempo de resposta, principalmente no caso de arquivos que quase nunca mudam, como bibliotecas e frameworks.

Por padrão, os navegadores já têm uma estratégia de cache, mas

podemos customizar a nossa.

Para ativarmos o cache customizado, precisamos adicionar um header em nossas respostas. Isso é bem simples, pois podemos criar um middleware que adiciona esse header em toda requisição.

```
app.use(function(req, res, next){
  var maxAge = 31557600000;
  res.setHeader('Cache-Control', 'public, max-age=' + maxAge/1
000);
  next();
});
```

Criamos o middleware, uma variável que representa um ano em milissegundos, e adicionamos o header Cache Control. Se você verificar no navegador, verá que ele já está lá.



Figura 4.3: Cache control

Apesar de ser um processo simples, o `express-static` – módulo que já usamos para servir arquivos estáticos – tem essa função pronta por baixo dos panos. Portanto, em vez de implementarmos esse middleware, podemos apenas passar um parâmetro avisando para ele o tempo de cache.

Agora, nossa chamada no `app.js` ficará assim:

```
app.use(static(path.join(__dirname, 'public'), {maxAge: 31557600})
);
```

Pronto, se verificar no navegador, o efeito deve ser o mesmo de

nosso `middleware` , porém, temos menos código para manter.

Só use seu próprio `middleware` caso queira implementar regras customizadas, como por exemplo, um cache para cada tipo de arquivo.

4.3 FAVICON

Nossa app está bem evoluída e cada vez rodando melhor em produção, mas esquecemos de algo importante, o Favicon.

Favicon é aquele ícone pequeno que fica ao lado do title da página em nossos navegadores. Ele também fica na barra de favoritos ao salvar um site, por isso o nome Favicon. Veja alguns exemplos:



Figura 4.4: Favicon

Vamos adicionar o módulo responsável por servir esse ícone, por meio do npm `npm install serve-favicon --save` .

Depois, vamos importar em nosso `app.js` :

```
//resto dos imports  
var favicon = require('serve-favicon');
```

Por fim, adicionamos o `middleware` em nossa pilha de `middlewares`:

```
app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
```

Neste caso, ele procura o arquivo `favicon.ico` em nossa pasta `public` . Procure por um, ou use o do projeto mesmo.

Ao recarregar o navegador, o ícone deverá estar lá:



Figura 4 5: Ícone

Se não aparecer, tente abrir a URL onde está o seu ícone (<http://localhost:3001/favicon.ico>) e, então, use o `hard refresh` (`cmd+shift+R` no Mac ou `shift+F5` em outras plataformas).

Isso ocorre pois os navegadores mantêm um cache muito forte no `favicons`, visto que eles quase nunca mudam.

4.4 NODE E THREADS

O Node.js nasceu para ser assíncrono. Esse framework baseado em eventos foi feito para escalar bem na web, porém existe um "problema" ao lidarmos com Node: ele só usa uma thread dos nossos processadores. A vantagem disto é que não temos problemas de concorrência, mas também não tiramos proveito dos múltiplos núcleos de nossos processadores.

Nos dias de hoje, é comum ver processadores com quatro ou oito núcleos, mas, por padrão, o Node usa apenas um.

Para resolver este problema, podemos tirar proveito dos **Clusters**. Um cluster nada mais é do que um outro processo Node. O nosso servidor vai decidir para qual dos processos a requisição web será redirecionada.

Para isto, vamos incluir o módulo `cluster`. Esse é um módulo nativo, e não precisamos instalar.

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

Precisamos também descobrir o número de núcleos de nosso processador para dispararmos uma quantidade de processos igual à quantidade de núcleos.

Poderíamos deixar esse número fixo também. Porém, e se um dia mudarmos de servidor, será que lembraríamos de alterá-lo? E cada desenvolvedor de nossa equipe pode ter uma quantidade diferente em seu computador, como lidar com isso?

Para pegar esse tipo de informação, existe o módulo `os`, que também é nativo.

```
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;
```

Agora que já temos o número de CPUs, precisamos definir um processo-pai, ou processo-mestre, que criará filhos de acordo com a quantidade de CPUs.

```
if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  };
}
```

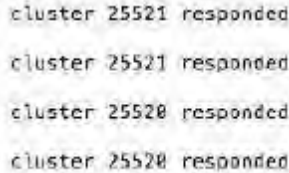
Neste código, verificamos se o processo é o mestre. Se for, criamos um `fork` – ou seja, uma cópia do processo – para cada número de CPUs. Agora, precisamos tratar os filhos.

```
else if (cluster.isWorker) {
  //fazemos algo
}
```

Nesse caso, vamos colocar todo o código de criação do servidor aqui dentro, ou seja, tudo que está em nosso `app.js`, com exceção dos imports. Vamos também criar uma rota para interceptar todas as requisições e criar um log de qual processo respondeu cada chamada:

```
app.get('*', function(req, res, next) {
  res.status(200);
  console.log('cluster '
    + cluster.worker.process.pid
    + ' responded \n');
  next();
});
```

Uma rota simples, que pega todas as requisições `get` e nos imprime qual processo respondeu.



```
cluster 25521 responded
cluster 25521 responded
cluster 25520 responded
cluster 25520 responded
```

Figura 4.6: Processos que responderam

O código completo do `app.js` :

```
require( './config/db' );

var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

var express      = require( 'express' );
var http         = require( 'http' );
var path         = require( 'path' );
var engine       = require( 'ejs-locals' );
var cookieParser = require( 'cookie-parser' );
var bodyParser   = require( 'body-parser' );
var methodOverride = require( 'method-override' );
var static       = require('serve-static');
var morgan       = require('morgan');
var fs           = require('fs');
var validator     = require('express-validator');
var compression  = require('compression');
var favicon      = require('serve-favicon');

if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  };
} else if (cluster.isWorker) {
  var app = express();
  app.use(favicon(__dirname + '/public/favicon.ico'));

  var routes = require( './routes' );

  app.get('*', function(req, res,next) {
```

```

        res.status(200);
        console.log('cluster '
            + cluster.worker.process.pid
            + ' responded \n');
        next();
    });

    app.set( 'port', process.env.PORT || 3001 );
    app.engine( 'ejs', engine );
    app.set( 'views', path.join( __dirname, 'views' ) );
    app.set( 'view engine', 'ejs' );
    app.use( methodOverride() );
    app.use( cookieParser() );
    app.use( bodyParser.json() );
    app.use( bodyParser.urlencoded({ extended : true }));
    app.use(validator());
    app.use( routes.current_user );
    app.get( '/', routes.index );
    app.post( '/create', routes.create );
    app.get( '/destroy/:id', routes.destroy );
    app.get( '/edit/:id', routes.edit );
    app.post( '/update/:id', routes.update );

    var env = app.get('env');
    app.use(compression({level: 9,memLevel: 9}));
    app.use(static(path.join( __dirname, 'public'),{maxAge: 3155760000
    0/1000}));

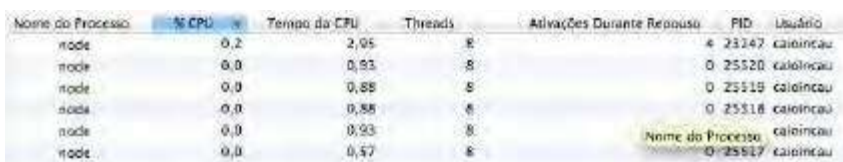
    if(env=="production"){
        var accessLogStream = fs.createWriteStream(__dirname + '/urls.
        log', {flags: 'a'})
        app.use(morgan('combined', {stream: accessLogStream}));
        require( './config/errors' )(app);
    }else{
        app.use(morgan('web'));
    }

    http.createServer( app ).listen( app.get( 'port' ), function (){
        console.log( 'Express rodando na porta : ' + app.get( 'port' ) )
    }
    );
};
};

```

Repare que a única diferença, além da rota que criamos para imprimir o processo, é que o código de criação do servidor, rotas e afins está dentro do `else`, ou seja, dentro dos processos-filhos. Deste modo, o processo-pai fica responsável apenas por decidir qual filho responde a cada requisição.

Outro modo de confirmarmos que há mais de um processo respondendo é abrir o monitorador de atividades ou o gerenciador de tarefas do seu sistema.



Nome do Processo	% CPU	Tempo da CPU	Threads	Ativações Durante Repouso	PID	Usuário
node	0,2	2,95	8		4 23247	caloimcau
node	0,0	0,93	8		0 25520	caloimcau
node	0,0	0,88	8		0 25519	caloimcau
node	0,0	0,88	8		0 25518	caloimcau
node	0,0	0,93	8			Nome do Processo caloimcau
node	0,0	0,57	8		0 25517	caloimcau

Figura 4.7: Gerenciador de tarefas

Veja que, mesmo com tantos processos, não está alto o consumo da CPU, pois a aplicação está em espera.

4.5 UMA OPÇÃO AO CLUSTER NATIVO

Se você prefere não ter de lidar com esse tipo de código, existe uma opção chamada `Cluster2`, criada e usada pelo eBay. Essa opção é mais aconselhável, principalmente se você não tem muito domínio sobre como os Clusters funcionam.

Criar um cluster com esse módulo é ainda mais simples do que criar com o módulo nativo.

Primeiramente, precisamos instalar o módulo em nossa aplicação:

```
npm instal cluster2 --save
```

Depois, fazemos os imports.


```
var Cluster = require('cluster2');
var express  = require( 'express' );
var http     = require( 'http' );
//resto dos imports
```

Agora, vamos realmente implementar o Cluster usando cluster2 . Ao utilizar esse módulo, não precisamos controlar se a Thread é Master ou Child , assim como fizemos ao usar o módulo nativo, pois o cluster2 cuidará disto automaticamente para nós.

Lembre-se de remover as alterações feitas para criar o Cluster com o módulo nativo (o `if / else` do `app.js`).

Ao final do `app.js` , no lugar de criarmos o servidor ao modo antigo:

```
http.createServer( app ).listen( app.get( 'port' ), function (){
  console.log( 'Express rodando na porta : ' + app.get( 'port' ) )
;
});
```

Vamos substituir a criação do servidor por uma que use o Cluster2 :

```
var c = new Cluster({
  port: 3000,
});
c.listen(function(cb) {
  cb(app);
});
```

Nesse código, ele inicia um novo cluster e pede para ele ouvir na porta 3000.

Suba o servidor e verifique se está tudo funcionando do mesmo modo.

Ao usar o `cluster2` , ainda ganhamos um painel para verificar em detalhes o estado da aplicação. Acesse <http://localhost:3001> e veja o painel, mostrado na figura a seguir:

Veda.local Darwin 13.3.0
Running since Sun Apr 19 2015 12:19:41 GMT-0300 (BRT)
PID: 37312
[Installed modules](#)

Master

Number of workers 4 (0 killed)
Number of core used 4 of 4
Mem usage at startup 1544.52 MB of 6144.00 MB
Current total mem usage 4784.6
CPU usage 17.28%
Average load 1.63 1.71 1.64

Logs

[Logs](#)

Counters

Worker 37332	
domain	null
_events	[object Object]
_maxListeners	10
_closesNeeded	2
_closesGot	0

Figura 4.8: Painel

Nele, você pode encontrar o ID do processo no seu sistema, o tempo que ele ficou de pé, o número de núcleos usados, quanta memória está consumindo e quanto está livre, e também os logs, que são todos os arquivos na pasta `/logs`, sendo assim, é uma boa prática jogarmos nossos logs lá.

Nos podemos customizar nosso Cluster com algumas opções:

1. `cluster` : recebe `true` ou `false` ; se for `false` , inicia com apenas um processo.
2. `port` : a porta que vamos usar em nossa aplicação; usamos 3000 em nosso caso.
3. `host` : o host usado para subir o servidor; a opção padrão é `0.0.0.0`
4. `monHost` : o host usado para a tela de monitoramento; usa o mesmo padrão do `host` .
5. `monPort` : a porta usada para a tela de monitoramento; por padrão, usa-se a 3001.
6. `noWorkers` : número de processos; por padrão, utiliza o `os.cpus().length` que aprendemos ao usar a biblioteca padrão `cluster` .

Veja um exemplo de uso com algumas customizações:

```
var c = new Cluster({  
  port: 3000,  
  cluster: true,  
  noWorkers: 2,  
  monPort: 3002  
});
```

Nesse caso, usamos apenas dois processos, e monitoramos a porta 3002. Poderíamos ainda omitir o `cluster:true` , pois este é seu valor padrão.

O `Cluster 2` é um módulo bem maduro e com ótima manutenção, sem dúvida a melhor opção para a maioria das aplicações Node que necessitam utilizar mais de um processo.

4.6 PROTEGENDO A APLICAÇÃO

CSRF

O CSRF (ou Cross-site request forgery) acontece quando um

cliente mantém a informação da sessão de um site protegido e, usando essa sessão, o servidor envia dados para o site da sessão original de forma maliciosa, como por exemplo, uma transferência de dinheiro.

O banco não sabe que essa requisição não veio de seu site, pois ele valida somente a sessão.

Para prevenir CSRF, nos podemos ativar uma proteção que usa um token e, a cada requisição, ele valida esse token.

A proteção contra CSRF pode ser feita com o módulo `csrf`, que adiciona um token chamado `_csrf` em nossa sessão (`req.session._csrf`) e valida-o. Caso o valor não seja válido, ele retorna um erro 403, que, como vimos, quer dizer que não encontrou o recurso esperado.

Vamos instalar o `csrf` em nossa aplicação (`npm install csrf --save`) e ativar esse módulo em nosso `app.js` :

```
var csrf = require('csrf');  
//resto app.js  
app.use(csrf({cookie:true}));
```

Lembre-se de que, como a ordem de execução dos nossos middlewares importam, precisamos sempre adicionar o `csrf` após a inicialização do `express()` e também do `cookie-parser` .

Repare que usamos a opção `cookie:true` . Isso indica para o `csrf` que usaremos cookies para controlar o token, e não a `session` .

Feitas as configurações iniciais, precisaremos adicionar o token em nossos formulários. Para isso, vamos sempre enviar o token para a view , em nosso arquivo `app.js` :

```
app.use(function(req, res, next) {  
  res.locals._csrf = req.csrfToken();  
  next();  
});
```

```
});
```

Repare que agora nosso objeto de `request` tem o método `csrfToken()`, que gera um token para nós.

Agora que temos acesso ao token na view, precisamos colocá-lo em nossos formulários. Temos dois formulários em nossa aplicação: um no `edit.ejs`, e outro no `index.ejs`

Veja o código do `edit.ejs`:

```
<% layout( 'layout' ) -%>

<h1 class="page-header"><%= title %></h1>

<% todos.forEach( function ( todo ){ %>
  <form action="/update/<%= todo._id %>" method="post" class="
form-create navbar-form navbar-left" accept-charset="utf-8">
    <div class="form-group">
      <input type='hidden' name='_csrf' value=<%= _csrf%> />
      Nome:<input required="required" class="form-control" type="
text" name="name" value="<%= todo.name %>" />
      Sobrenome:<input required="required" class="form-control" t
ype="text" name="lastName" value="<%= todo.lastName %>" />
      Tarefa:<input required="required" class="form-control" type=
"text" name="content" value="<%= todo.content %>" />
    </div>
    <input type="submit" value="Gravar" class="btn btn-default">
    <a href="/destroy/<%= todo._id %>" title="Delete esse item" c
lass="btn btn-default">Delete</a>
  </form>

<% }); %>
```

Adicione também o `hidden input` no `index.ejs`:

```
<% layout( 'layout' ) -%>

<h1 class="page-header"><%= title %></h1>

<div id="list">
  <form action="/create" method="post" accept-charset="utf-8" clas
s="navbar-form form-create">
    <div class="form-group">
      <input type='hidden' name='_csrf' value=<%= _csrf%> />
```

<%#Resto do index.ejs %>

Por padrão, o `csrf` não valida o token em requisições do tipo GET ; o que faz sentido, visto que o GET não deveria alterar nada em nosso servidor.

Header HTTP de segurança

O protocolo HTTP é extremamente extensível e, com o passar dos anos desde sua implementação, foram criados inúmeros tipos de headers.

Alguns desses headers são usados para dificultar a exploração de vulnerabilidades de uma aplicação. Por contribuir com essa proteção, conhecê-los é muito importante.

Existe um módulo chamado `helmet` , que nos fornece um modo simples de adicionar a maioria dos headers de segurança, sendo eles:

- `contentSecurityPolicy` : este header serve para especificar de quais domínios podemos carregar arquivos em nosso site.
- `hidePoweredBy` : remove o header `X-Powered-By` . Hoje, se abrimos o Inspetor no Chrome, podemos ver que nossa aplicação usa Express.



Figura 4 9: X-Powered-By

Isso pode fazer com que pessoas mal-intencionadas abusem de bugs conhecidos de determinados frameworks.

```
app.use(helmet.hidePoweredBy());
```

- `hpkp` : para HTTP Public Key Pinning, certificados HTTPS podem ser forjados, permitindo ataques em apps que usam SSL. HTTP Public Key Pinning tenta evitar que isto ocorra.
- `hsts` : evita que os usuários acessem seu site por meio de HTTP, quando existe a opção de usar HTTPS (SSL ativado).

```
app.use(helmet.hsts({ maxAge: 7776000000 }));
```

- `ieNoOpen` : muda o header `X-Download-Options` para `noopen`, prevenindo que usuários de IE executem HTMLs baixados em nosso site.

Algumas aplicações web servem HTML não confiável para download. Por padrão, algumas versões do IE permitem abrir esses HTMLs no contexto do seu site, o que permite executar código malicioso nele.

```
app.use(helmet.ieNoOpen());
```

Você pode encontrar detalhes sobre este bug em um post da Microsoft, disponível em <http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx>.

- `noCache` : como costumamos usar caches com grande período de expiração, se colocarmos em produção uma aplicação com bug em nosso HTML ou JavaScript, e pouco tempo depois colocarmos uma nova versão arrumando, o usuário poderá continuar acessando a aplicação com problemas.

```
app.use(helmet.noCache());
```

Utilizando o `noCache` , podemos invalidar o cache do usuário.

- `noSniff` : coloca o header `X-Content-Type-Options` com a opção `nosniff` .

```
app.use(helmet.noSniff());
```

Um sniff é o ato de rodar um arquivo com extensão errada, como por exemplo, `<script src="danger.txt"></script>`

Alguns browsers tentariam interpretar esse `txt` como um arquivo `js` .

- `frameguard` : adiciona o header `X-Frame-Options` para evitar o Clickjacking (ou roubo de click), uma técnica que consiste em adicionar um elemento invisível em cima de algum elemento que o usuário pretende clicar, fazendo assim, com que ele clique em algo malicioso.

O `frameguard` aceita três opções: `DENY`, `SAMEORIGIN`, e `ALLOW-FROM`, sendo que ao usar `DENY`, a página não pode ser mostrada em um `iframe`, impedindo que usem sua página em clickjackings; já o `SAMEORIGIN` permite que sua página seja mostrada como `iframe`, mas somente no mesmo domínio da página; e, por último, o `ALLOW-FROM`, que recebe um segundo parâmetro especificando qual URI pode acessar sua página por meio de `iframes`.

```
app.use(helmet.frameguard('allow-from', 'http://example.com'));
```

Se você não passar nenhum parâmetro, ele usará o `SAMEORIGIN` como padrão.

- `xssFilter` : adiciona o header `X-XSS-Protection` , tentando evitar ataques XSS.

```
app.use(helmet.xssFilter());
```


Ataques XSS ocorrem quando alguém consegue adicionar um script malicioso na página de algum modo, como por exemplo, inserindo um `import JavaScript` em um campo de texto. Infelizmente, esse header ainda é suportado apenas pelo IE9+ e Google Chrome.

Configurando o helmet

Agora que entendemos as principais funcionalidades do `helmet`, vamos instalá-lo em nossa aplicação:

```
npm install helmet --save
```

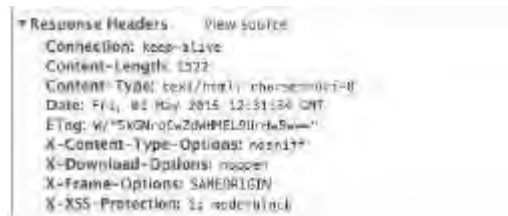
E adicionar o módulo em nosso `app.js`:

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

```
app.use(helmet());
```

Por padrão, o `helmet` já deixa ativado toda as opções, exceto `contentSecurityPolicy`, `hpkp` e `noCache`, o que para nosso caso é mais do que o suficiente.

Se abrirmos o inspetor do Chrome, veremos nossos novos headers por lá:



4.7 REVISANDO

Neste capítulo, vimos como comprimir e ativar o cache em nossos arquivos servidos para o cliente, como implementar um

cluster usando apenas as bibliotecas nativas, e também como criar a partir do módulo `cluster2` .

ENVIO DE E-MAILS COM NODE.JS

Uma tarefa muito comum nas aplicações é o envio de e-mail, sendo assim, vamos aprender a usar o envio de e-mail por meio do módulo `nodemailer`.

Vamos instalar o módulo:

```
npm install nodemailer --save
```

Em vez de adicionarmos em nosso `app.js`, vamos criar um arquivo para configurar nossos e-mails, dentro da pasta `config`. Criaremos o arquivo `mailer.js` que será responsável pelas configurações e envio de e-mails em nossa aplicação.

Vamos começar importando o `nodemailer`:

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

Depois disso, precisamos configurar um `transporter`, que nada mais é que um objeto capaz de realizar o envio de e-mails. Para isso, vamos usar o método `createTransport` do `nodemailer`:

```
var nodemailer = require('nodemailer');
module.exports = function() {
  var transporter = nodemailer.createTransport({
    service: 'Gmail',
    auth: {
      user: 'SEUGMAIL@gmail.com',
      pass: 'SUASENHA'
    }
  })
}
```

```
});  
};
```

Em nosso exemplo, vamos usar o Gmail. Repare que o método recebe o serviço que usaremos, neste caso, Gmail e os dados para autenticar. Não se esqueça de substituir esses dados por uma conta válida do Gmail.

Se você realmente quiser enviar o e-mail, precisará permitir que aplicações menos seguras possam enviar e-mail por meio da sua conta do Gmail. Isso indica que você permite que outras aplicações acessem sua conta sem uso de tokens e afins.

Para ativar, basta acessar <https://www.google.com/settings/security/lesssecureapps>. Você receberá um e-mail confirmando esta ativação.

Sugiro usar um e-mail novo para evitar problemas com sua conta atual, já que durante os testes, o envio de e-mails pode acabar lhe incomodando.

Configurado isso, vamos agora configurar o e-mail:

```
var nodemailer = require('nodemailer');  
module.exports = function() {  
  var transporter = nodemailer.createTransport({  
    service: 'Gmail',  
    auth: {  
      user: 'SEUGMAIL@gmail.com',  
      pass: 'SUASENHA'  
    }  
  });  
  
  var mailOptions = {  
    from: 'Seu Nome <SEUGMAIL@gmail.com>',  
    to: 'SEUGMAIL@gmail.com, SEUSEGUNDOEMAIL@gmail.com',  
    subject: 'Tarefa Criada',  
    html: '<b>Tarefa Criada </b>'  
  };  
};
```

A configuração é algo bem simples. Nela configuramos quem

vai enviar (`from`), e o padrão é o nome seguido por `<email>` . Em nosso caso, ficou:

```
from: 'Seu Nome <SEUGMAIL@gmail.com>'
```

Depois, configuramos quem receberá (`to`). Repare que separei mais de um e-mail por vírgula; você pode passar um ou mais e-mails separados dessa maneira:

```
to: 'SEUGMAIL@gmail.com, SEUSEGUNDOEMAIL@gmail.com'
```

Em seguida, temos o assunto do e-mail (`subject`). Neste caso, notificaremos o administrador (somos nós mesmos) sobre a criação de novas tarefas no sistema:

```
subject: 'Tarefa Criada'
```

Por fim, temos o corpo do e-mail (`html`), onde há todo o conteúdo do seu email . O `nodemailer` aceita envio de e-mails no formato HTML. Veja que usamos a tag bold (``) para deixar o texto em negrito:

```
html: '<b>Tarefa Criada </b>'
```

Por fim, vamos criar a função que realmente envia o e-mail, utilizando a função `sendMail` de nosso `transporter` . Esta recebe as opções que criamos (`mailOptions`) e uma função de callback que nos informa o resultado da operação, podendo ser um erro ou apenas a informação de sucesso:

```
var sendMail = transporter.sendMail(mailOptions, function(error, info){
    if(error){
        console.log(error);
    }else{
        console.log('Email enviado: ' + info.response);
    }
});
```

Neste caso, estamos apenas logando no console, tanto sucesso quanto erro, mas poderíamos tentar enviar e-mail de outro modo,

ou avisar o administrador através de outra ferramenta, como o **hipchat**. Você pode conhecê-lo em <https://www.hipchat.com/>.

O nosso `mailer.js` ficará assim:

```
var nodemailer = require('nodemailer');
module.exports = function() {

  var transporter = nodemailer.createTransport({
    service: 'Gmail',
    auth: {
      user: 'SEUGMAIL@gmail.com',
      pass: 'SUASENHA'
    }
  });

  var mailOptions = {
    from: 'Seu Nome <SEUGMAIL@gmail.com>',
    to: 'SEUGMAIL@gmail.com, SEUSEGUNDOEMAIL@gmail.com',
    subject: 'Tarefa Criada',
    html: '<b>Tarefa Criada </b>'
  };

  var sendMail = transporter.sendMail(mailOptions, function(error, info){
    if(error){
      console.log(error);
    }else{
      console.log('Email enviado: ' + info.response);
    }
  });
};
```

Vamos querer enviar esse e-mail toda vez que uma tarefa for criada. Para isso, vamos ao nosso `index.js` e importaremos o nosso `mailer`.

```
var utils = require( '../utils' );
var mongoose = require( 'mongoose' );
var Todo = mongoose.model( 'Todo' );
var mailer = require('../config/mailer');

//resto do index.js
```

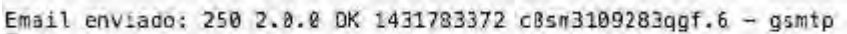
Agora, em nosso método `create`, vamos efetivamente chamar a

função que envia o e-mail:

```
exports.create = function ( req, res, next ){
  req.checkBody('name', 'Nome é necessário').notEmpty();
  errors = req.validationErrors();
  mailer();

  //resto da função
}
```

Veja em seu console que ele logou o envio de e-mail:

A terminal window showing the output of an email sending process. The text reads: "Email enviado: 250 2.0.0 DK 1431783372 c8sn3109283qgf.6 - smtp".

Email enviado: 250 2.0.0 DK 1431783372 c8sn3109283qgf.6 - smtp

Figura 5 1: Email enviado

Também pode conferir em seu Gmail que o envio foi um sucesso:



Figura 5 2: Email enviado

5.1 MANTENDO SUA APLICAÇÃO DE PÉ

É muito importante garantir disponibilidade de sua aplicação em produção. Para isso, vamos usar um módulo chamado `forever`, que faz com que sua aplicação Node.js fique de pé sempre; se o processo por algum motivo morre, ele sobe novamente.

```
sudo npm install -g forever
```

Lembre-se de usar o `-g`, afinal, esse será um módulo de acesso global.

O uso do `forever` é bem simples, basta rodarmos nosso `app.js` com ele:

```
forever start
```

Pronto, assim, se seu processo morrer, ele o subirá novamente. Mas é muito importante lembrar de que **isso não vai funcionar caso seu servidor reinicie**.

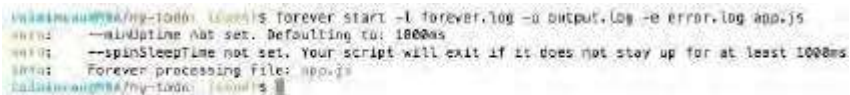
Agora que vimos o básico, vamos nos aprofundar nesse módulo.

Uma das funções mais interessantes do `forever` é manter seu terminal livre. Assim, vamos redirecionar os logs para arquivos separados, algo muito comum em produção:

```
forever start -l forever.log -o output.log -e error.log app.js
```

Com a opção `-l`, vamos fazer com que os logs do próprio `forever` vão para o arquivo indicado a seguir. O `-o` redireciona os logs padrões da aplicação (`stdout`), e o `-e` redireciona os logs de erro (`stderr`).

Suba sua aplicação e verifique que o terminal ficou livre.



```
calvin@calvin:~/my-1000s$ forever start -l forever.log -o output.log -e error.log app.js
forever: --minUptime 100s set. Defaulting to: 1000ms
forever: --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
forever: Forever processing file: app.js
calvin@calvin:~/my-1000s$
```

Figura 5.3: Terminal livre

Mas e agora, como pararemos de rodar o nosso servidor? Temos duas alternativas: matar todos os processos que o `forever` está gerenciando, por meio do `stopall`:

```
forever stopall
```

Ou parar somente nosso processo com `stop <arquivo>`:

```
forever stop app.js
```

Verifique que o comando funciona. Veja que ele nos mostra algumas informações, como o tempo em que a aplicação ficou de pé e para onde foram os logs.


```

calvin@kali:~/my-todo $ npm run forever stop app.js
info: Forever stopped process:
      command      script      pid      exit    logfile      uptime
[0] ylnf /usr/local/bin/node app.js 15177 15186 /Users/calvin/.forever/forever.log 0:0:13.348

```

Figura 5.4: forever stop app.js

Assim como temos as opções `stop` e `stopall`, temos funções análogas para fazer reiniciar nosso processo:

```

forever restart app.js
forever restartall

```

Tente subir novamente o servidor redirecionando os logs para os mesmos arquivos de antes:

```

forever start -l forever.log -o output.log -e error.log app.js

```

Você deverá receber um erro:

```

calvin@kali:~/my-todo $ npm run forever start -l forever.log -o output.log -e error.log app.js
info: ---minimize not set. Defaulting to: 1000ms
info: ---spinGleecTime not set. Your script will exit if it does not stay up for at least 1000ms
info: Forever processing file: app.js
error: Cannot start forever:
error: log file /Users/calvin/.forever/forever.log exists. Use the -a or --append option to append log.

```

Figura 5.5: Erro

Este erro ocorre, pois o arquivo `forever.log` já existe, e nós não podemos sobrescrevê-lo, além de não ser interessante, pois iríamos perder logs.

Para resolvermos isso, existe a opção de passarmos a flag `-a`. Esta indicará para o `forever` que ele deve fazer `append` nos arquivos de log, e não `sobrescrita`, ou seja, todo o conteúdo novo será somado ao arquivo.

```

forever start -l forever.log -o output.log -e error.log -a app.js

```

Se você quiser garantir que sua aplicação está de pé, pode conferir tanto no navegador, acessando-a normalmente, quanto usando o comando `forever list`.

[illegible]

Figura 5.6: forever list

Este comando é interessante, pois podemos ver todas as aplicações que o `forever` está gerenciando e seu tempo de pé (`uptime`).

Repare que ele também nos devolve o `pid`, ou seja, o ID do processo. Com ele, podemos derrubar a aplicação por meio do `forever stop`, passando como argumento o `pid`:

No exemplo da figura anterior, seria:

```
forever stop 16246
```

Isto é equivalente a `forever stop app.js`, com a vantagem de que não precisamos nos preocupar em estar na pasta que o script está ou passar o caminho até ele.

Para ver uma lista com mais opções, você pode usar o `forever -help`, que vai mostrar uma lista extensa.

5.2 MIGRANDO DO EXPRESS 3 PARA O 4

Neste livro, nós usamos o Express 4. Apesar de sua grande adoção, essa versão foi lançada no final de 2014 e nem todas as aplicações já a usam.

Por isso, vamos apresentar as principais diferenças e como você deve proceder para realizar a migração de seu projeto com Express 3 para o 4.

Remoção de módulos padrões.

A principal diferença foi a remoção de alguns módulos que

faziam parte do Express. Foram removidos os módulos `bodyParser` , `cookieParser` , `favicon` e `session` .

Para fazer uso deles, você agora precisa importá-los em seu `app.js` , como fizemos durante o livro.

```
var cookieParser = require( 'cookie-parser' );
var bodyParser   = require( 'body-parser' );
var favicon      = require('serve-favicon');
```

O único que não usamos durante o livro foi o `session` , que pode ser importado do mesmo modo:

```
var session = require('express-session');
```

Métodos removidos do core

Nós tivemos alguns métodos removidos direto do `core` (`express()`). Precisamos ficar atentos para alterar/ remover o uso destes.

O primeiro deles é o `app.configure()` , que recebia um `environment` e uma função que só era executada neste `environment`.

```
app.configure('development', function() {
  // faz algo
});
```

Para substituí-lo, precisamos fazer um `if` para verificarmos o `environment`. Também fizemos isso durante o livro:

```
var env = process.env.NODE_ENV || 'development';
if ('development' == env) {
  // faz algo aqui
}
```

O segundo é o `app.router` , que servia para definir a ordem na qual as rotas e `middlewares` eram executados:

```
app.use(app.router);
```

Essa linha precisa ser removida, pois agora as rotas e os

middlewares são sempre definidos/ executados na ordem em que são adicionados a seu arquivo de configuração das rotas, em nosso caso, no `app.js`

Por último, mas não menos importante, o próprio método que inicializa o Express foi alterado, passando de `express.createServer()` para apenas `express()` .

Métodos do core que foram alterados

Alguns métodos não foram removidos, mas apenas alterados para melhorar seu uso. O primeiro deles foi o `app.use()` , que agora aceita URLs parametrizadas:

```
app.get( '/destroy/:id', routes.destroy );
```

Veja que já usamos em nosso projeto, neste caso, o `:id` , que é um parâmetro.

O método `req.accepted()` também foi alterado para `req.accepts()` , seu uso continua o mesmo.

O método `res.location()` não é mais necessário. Ele servia para resolver URLs relativas, mas o próprio browser cuida disso.

Mudanças de configuração

Foram feitas algumas mudanças menos significativas na parte de configuração. O `req.params` antes era um array, agora passou a ser um objeto, ou seja, agora você pode acessar seus parâmetros por meio de `req.params.nomeDoParametro` , sendo que antes eram acessados exclusivamente através da chave `req.params["nomeDoParametro"]`

O `res.locals` também virou um objeto em vez de uma função.

A maioria das mudanças ocorreram devido à remoção da

dependência Connect (<http://senchalabs.github.com/connect/>) do Express.