Casa do Código
Livros para o programador
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Prefácio

É impressionante a quantidade de conhecimento que o Flávio colocou nesse livro. Para mim, esse é o melhor resumo do que você está por encontrar nas próximas páginas.

Você vai aprender uma solução ponta a ponta completíssima usando JavaScript no front-end com Angular, no back-end com Node e Express e no banco de dados com MongoDB. Mas não só. Você vai ver testes de unidade e integração contínua; vai ver build de front-end com Grunt.js e até fazer deploy da aplicação.

O conteúdo me impressiona porque cobre todo o desenvolvimento, apresentando ferramentas para todas as etapas. Isso é muito raro em um livro só. Ao mesmo tempo, o livro é enxuto e *na medida certa*. Todo o essencial está aqui, mas sem passar do ponto e ser extenso demais.

Se você trabalha com Web, recomendo fortemente o livro. A MEAN Stack com JavaScript em todas as pontas é interessantíssima. Abre muitas possibilidades. Mas vou além e digo que você vai aprender muita coisa aqui, mesmo se não for usar alguma parte do MEAN. Talvez sua aplicação não precise de Angular; mas o conhecimento de Express e Mongo pode-lhe ser útil. Ou talvez você vá usar outro banco de dados, mas vai aprender muito de Angular, Grunt e testes aqui.

Conheço o Flávio há alguns anos já, trabalhando juntos na Caelum. Seu trabalho sempre é pautado por muita seriedade, fundamentação e uma didática incrível. Sempre vi isso em suas aulas e palestras. E agora vejo essas boas características ainda mais acentuadas neste livro. É uma excelente obra, garanto que você vai aproveitar. Boa leitura!

Sérgio Lopes

Agradecimentos

Agradeço a Sérgio Lopes por ter reservado parte do seu escasso tempo para a leitura e elaboração do prefácio deste livro. Aproveito também para agradecer à Casa do Código, mais notadamente ao Adriano Almeida e Vivian Matsui pelo pro□ssionalismo e trato despendidos durante todas as etapas de criação deste livro.

Por □m, não poderia deixar de agradecer à minha esposa Camila Almeida pelo apoio durante os nove meses de gestação deste livro, especialmente por compreender a importância do meu silêncio durante as revisões matinais e noturnas realizadas durante a nossa travessia Niterói/Rio.

Sumário

CZ£ –¶ 1

Introdução

"O único homem educado é o homem que aprendeu a aprender"
– Carl Rogers

Alguma vez você já precisou:

- Debugar sua aplicação no navegador, no *server* e no banco de dados?
- Realizar *migrations* toda vez que seu esquema de banco mudar?
- Criar um protótipo para rápida validação com o cliente?
- Gerenciar as dependências de seu projeto, tanto do servidor quanto do cliente?
- Automatizar o processo de *build* de sua aplicação?

- Consumir *REST Endpoints* diretamente na *view*?
- Traduzir consultas relacionais em objetos e objetos para o mundo relacional?
- Manipular o DOM adicionando, removendo e alterando elementos dinamicamente?
- Realizar testes unitários e *end-to-end*?
- Realizar o *deploy* da aplicação?
- Criar componentes reutilizáveis na *view*?

Pode parecer um tanto estranho para quem vem procurar respostas em um livro se deparar com uma série de perguntas logo de início.

Agora pense nas linguagens e frameworks que você já utilizou para suprir as necessidades listadas. Pensou? Você provavelmente deve ter chegado a milhões de frameworks e diferentes linguagens, certo? E se eu lhe dissesse que é possível supri-las com apenas uma linguagem, ou melhor, com um quarteto que fala a mesma língua? Seja bem-vindo à MEAN Stack.

1.1 MEAN S□ Zh□

O acrônimo MEAN foi cunhado em 2013 por Valeri Karpov do time do MongoDB [3] para denotar o uso de uma *stack* completa para desenvolvimento de aplicações incluindo MongoDB, Express, AngularJS e Node.js:

(M)ongoDB

MongoDB (http://www.mongodb.org) é um poderoso banco de dados NoSQL □exível e escalável. Combina a habilidade de escalar com os muitos recursos presentes nos bancos de dados relacionais como índices, ordenação etc. [1].

Na MEAN Stack, MongoDB permite armazenar e reaver um dado em um formato muito similar ao JSON (*JavaScript Object Notation*) que veremos mais à frente. É um banco fracamente tipado, o que ajuda muito pouco com

validação de dados, de modo que boa parte da responsabilidade das regras de validação □ca nas mãos dos desenvolvedores.

(E)xpress

Express (http://expressjs.com) , criado em 2009 por TJ Holowaychuk, é um framework web *light-weight* que ajuda na organização de sua aplicação web na arquitetura MVCno lado do servidor. Você pode usar uma série de templates engines como ejs, jade, hogan. Apesar desses engines, na MEAN Stack, o Express geralmente disponibiliza REST Endpoints (http://pt.wikipedia.org/wiki/REST) que são consumidos pelos templates do AngularJs, que por sua vez são atualizados com base nos dados recebidos.

(A)ngularJS

AngularJS (http://angularjs.org/) é um framework MVC no lado do cliente voltado para *Single Page Web Applications* (SPA) criado pela Google e liberado para o público em 2009. Ele simpli□ca bastante o desenvolvimento, manutenção e testes. Diferente da maneira tradicional, na qual dado e lógica são processados no servidor, AngularJS facilita o recebimento de dados e execução de lógica diretamente no cliente.

Na MEAN Stack, AngularJS é o responsável por interfaces dinâmicas sem manipulação de DOM e permite acessar facilmente REST Endpoints diretamente do cliente através de serviços especializados.

(N)odeJs

Node.js (http://nodejs.org) é uma plataforma para aplicações JavaScript criada por Ryan Dahl sob o ambiente de execução JavaScript do Chrome. É possível utilizar bibliotecas desenvolvidas pela comunidade através de seu gerenciador de pacotes chamado npm.

Na MEAN Stack, Node.js é a condição *sine qua non* para seu funcionamento, o core.

1.2 U Z Z Z JZ S h

MEAN aposta na onipresença da linguagem JavaScript, o que signi ca que a comunidade pode contribuir com novas stacks baseadas em JavaScript na plataforma Node.js que tornem ainda mais produtivo e divertido o desenvolvimento de aplicações web [4]. Um exemplo disso é a *fullstack* JavaScript Meteor (https://www.meteor.com) , também baseada no Node.js e no MongoDB.

1.3 VZ Z u oZ Zh

Podemos facilmente pensar que, por utilizarmos quatro tecnologias distintas, teremos mais trabalho na implementação e que o desenvolvimento será menos produtivo, certo? Na verdade, para provar o contrário, Karpov utilizou a stack em competições de Hackathon, que são curtas competições de programação, provando o quanto era produtiva. Devido a esta produtividade, a stack foi utilizada em um de seus projetos comerciais, chamado Ascot Project (http://www.ascotproject.com) , uma ferramenta de *image-tagging* para marcar produtos em fotos. É possível ainda listar essas vantagens:

Integração da equipe

A onipresença da linguagem ajuda na integração da equipe: front-end e back-end trabalhando com a mesma linguagem! Isso diminui o *truck factor*.

Debugging

Debugar a aplicação torna-se mais fácil quando os objetos armazenados no banco de dados são praticamente semelhantes aos objetos no lado do cliente. Por exemplo:

```
// no navegador
console.log(contato)
{
    "_id":    "5303e0649fd139619aeb783e"
    "nome":  "Flávio Almeida"
}
```

```
// recebido no servidor
console.log(contato)
{
      "_id": "5303e0649fd139619aeb783e"
      "nome": "Flávio Almeida"
}

// buscado do banco de dados
console.log(contato)
{
      "_id": ObjectId("5303e0649fd139619aeb783e")
      "nome": "Flávio Almeida"
}
```

Repare que o último exemplo devolve um *ObjectId*, que veremos mais à frente quando abordarmos o MongoDB.

Prototipação

Protótipos ajudam a validar ideias, tomar decisões e descobrir novas possibilidades. Com MEAN, desenvolvedores conseguem protótipos rápidos, já que declaram o modelo como POJO (*Plain Old JavaScript Object*) no AngularJS, este último utilizado o servidor Express apenas como ponte de acesso ao MongoDB. A estrutura do dado enviado é semelhante no cliente, no servidor e no banco, o que reduz drasticamente o tempo com conversões.

Performance

Há a possibilidade, quando necessária, de processar totalmente ou parcialmente dados e lógica no cliente, onerando menos o servidor.

Build, teste e gerenciamento de dependências

Não é necessário utilizar ferramentas de build ou de gerenciamento de pacotes front-end em outras linguagens, tudo pode ser feito com JavaScript. Veremos como utilizar Grunt (http://gruntjs.com) para automatizar tarefas e Bower (http://bower.io) para gerenciar nossos pacotes de front-end.

Mas como vamos juntar todas essas peças e construir algo rapidamente? Quais passos seguiremos? É o que veremos a seguir.

1.4 V[illegible]

O ecossistema em torno do JavaScript continua a evoluir. A MEAN Stack é uma tentativa de aproveitá-lo, mas não é um [illegible]m em si. Há aqueles que utilizam-na apenas para prototipação, outros por curiosidade ou em aplicações comerciais. Este livro não se coaduna com nenhum desses usos: cabe ao leitor decidir como empregará a stack, mas para isso ele precisa estar informado.

O livro fornecerá uma visão geral da stack no contexto de um CRUD de contatos chamado **Contatooh**. A instalação do Node.js será nosso ponto de partida. Em seguida, entraremos no Express 4 e aprenderemos como expor REST Endpoints através de rotas. Nesta etapa, não teremos integração com banco, apenas dados voláteis na memória do servidor serão retornados.

Na sequência, passaremos para o AngularJS. Primeiro, vamos aprender seus conceitos fundamentais, para depois integrá-lo com o servidor Express, consumindo os REST Endpoints que foram criados. Aproveitaremos esta etapa para aplicar com zero de esforço um visual pro[illegible]ssional à aplicação através do Bootstrap, concluindo a interface da aplicação. Todas as dependências front-end serão baixadas e gerenciadas pelo Bower.

Depois, no lugar de retornarmos e atualizarmos dados voláteis, integraremos nosso back-end Express com o MongoDB através do Mongoose, uma biblioteca que reduz signi[illegible]cantemente a quantidade de código utilizada nesta integração.

Ativaremos um sistema de autenticação utilizando o protocolo OAuth 2.0, o que permitirá que apenas usuários autenticados tenham acesso ao sistema. Neste momento, já teremos todas as funcionalidades do projeto prontas.

Com todas as funcionalidades do projeto estabelecidas, automatizaremos tarefas de otimização da aplicação através do Grunt. Vamos criar testes unitários utilizando o Karma/Jasmine e testes end-to-end (e2e) com Protractor.

Integraremos nossa aplicação com o Travis, um servidor de integração

contínua gratuito para projetos open source. Inclusive, vamos aprender a rodar nossos testes *e2e* na nuvem, integrando o Travis com o Sauce Labs, um TaaS(*test as a service*) também gratuito para projetos open source. Em seguida, vamos ver como criar e testar nossos próprios componentes de view reutilizáveis através das diretivas do AngularJS.

Por fim, realizaremos o deploy da nossa aplicação no OpenShift, um PaaS (*platform as a service*) que permite o deploy gratuito em seu *public PaaS* . O deploy será realizado pelo Travis, nosso servidor de integração, apenas se a construção do projeto for bem-sucedida e se todos os testes tiverem passado; isso permite que tenhamos, para cada commit da aplicação, um deploy!

Por que não usar um gerador?

Existem na comunidade geradores que fazem *scaffold* da MEAN Stack que aceleram bastante o tempo de desenvolvimento. Porém, o autor preferiu construir a stack com o leitor, passo a passo, para que você compreenda cada tecnologia individualmente ao mesmo em que integra cada uma delas ao longo do livro.

Não esgotaremos cada tecnologia da composição MEAN, mas o leitor terá a base para se aprofundar ainda mais em cada uma delas. A Casa do Código possui obras que podem ajudá-lo nesta tarefa:

- Aplicações web real-time com Node.js
- REST: Construa API's inteligentes de maneira simples

Da mesma maneira que um prédio precisa de uma fundação antes de ser construído, a fundação do nosso projeto será o Node.js. Veremos como instalá-lo a seguir.

1.5 Instalando o Node.js

A instalação do Node.js é tão simples quanto clicar no botão Install em seu website (http://nodejs.org) . O navegador inteligentemente baixará a versão para sua plataforma, seja ela Linux, Mac ou Windows.

Com o arquivo de instalação em mãos, basta executá-lo e seguir o assistente de instalação até o fim. É possível verificar se tudo ocorreu bem executando no terminal o comando *node -v* na linha de comando.

Linux: node ou nodejs

Nas distribuições Ubuntu e derivadas, pode haver um conflito de nomes quando o Node é instalado pelo apt-get. Neste caso específico, no lugar do binário se chamar node, ele passa a se chamar **nodejs**. Se o seu sistema é uma dessas distribuições, não se preocupe, basta trocar a chamada node por nodejs.

Se tudo estiver funcionando, será exibida no console a versão do Node.js instalada:

```
node -v
v0.10.33
```

Versões do Node.js superiores a 0.6.2 possuem por padrão o npm instalado, o que facilita ainda mais sua instalação.

CZ£ –¶ 2

Express: framework web para Node.js

"Bem começado, metade feito."
– Aristóteles

Vimos na introdução que o Express (http://expressjs.com) é um framework web *light-weight* criado em 2009 por TJ Holowaychuk que ajuda na organização de sua aplicação web, na arquitetura MVC no lado do servidor. Qual a razão de sua existência, uma vez que o Node.js já possui a capacidade de servir conteúdo na web? Vejamos um exemplo prático, mas sem nos preocuparmos com o código por enquanto.

Crie um arquivo chamado **node-server.js** com o seguinte código:

```
var http = require('http');
http.createServer(function (req, res) {
```

```
  res.writeHead(200, {'Content-Type' : 'text/plain'});
  res.end('Sou um servidor criado pelo Node.js!\n');
}).listen(3000, '127.0.0.1');
```

Agora, rode o script no terminal através do comando node:

```
node node-server
```

Para testar, abra a URL (http://localhost:3000) em seu navegador de preferência. Ele deve exibir a mensagem:

```
Sou um servidor criado pelo Node.js!
```

Apesar de funcional, nosso servidor é limitado. E se agora quisermos acessar um recurso a partir de uma URL diferente? Se quisermos usar templates? A lista de necessidades pode ser maior. É aí que entra o Express.

2.1 E□□u □□ u □u □□Z□u □

O Express estende as capacidades do servidor padrão do Node.js adicionando **middlewares** e outras capacidades como views e rotas.

Middlewares

Middlewares são funções que lidam com requisições. Uma pilha de middlewares pode ser aplicada em uma mesma requisição para se atingir diversas □nalidades (segurança, logging, auditoria etc.). Cada middleware passará o controle para o próximo até que todos sejam aplicados.

Revendo o código anterior, substituiríamos a função de callback passada para a função http.createServer por uma instância do Express, que age como um request listener aplicando sua pilha de middlewares.

```
http.createServer('Aqui entra o Express com seus middlewares')
    .listen(3000, '127.0.0.1');
```

E[[u [[3.X u C[[[u h[

Até a versão 3.X o Express utilizava o Connect (http://www.senchalabs.org/connect) . Ele foi criado para estender as capacidades do servidor HTTP do Node.js provendo middlewares de alta performance. Agora, cada middleware do Express é um módulo em separado que precisa ser instalado através do npm. Com esta mudança, cada módulo poderá receber correções e atualizações sem interferir nas versões do Express.

Agora que o papel do Express já foi esclarecido, podemos partir para a criação do projeto.

2.2 E[[[[[[[Z o[[[[[u [[u [Zh[Z[u .[[[[

Nosso primeiro passo será criar o diretório contatooh que representa nosso projeto, inclusive com subpastas para ajudar a separar e agrupar nosso código:

```
contatooh
  app
    controllers -> controladores chamados pelas rotas da
                   aplicação
    models -> models que representam o domínio do problema
    routes -> rotas da aplicação
    views -> views do template engine
  config -> configuração do express, banco de dados etc.
  public -> todos os arquivos acessíveis diretamente pelo
            navegador
```

O próximo passo será criar o arquivo package.json na raiz do projeto.

package.json

O arquivo package.json possui informações sobre o projeto (autor, nome, versão etc.) e uma lista com todas as dependências da nossa aplicação, isto é, os módulos de que dependemos. Ele utiliza a estrutura JSON (*JavaScript Object Notation*) familiar a programadores JavaScript.

Poderíamos até criar na mão o arquivo package.json, mas utilizaremos o comando npm init:

```
npm init
```

Um assistente fará uma série de perguntas sobre o projeto. Podemos dar ENTERpara todas até que o assistente termine. No nal, teremos um package.json com o nome do projeto igual à pasta onde executamos o npm:

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane
defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package
and save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (x)
(ENTER para todas as perguntas)
```

Quando o assistente chegar ao nal ele terá criado o arquivo package.jon com a seguinte estrutura:

```
{
  "name": "contatooh",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

dentro dela temos o Express que acabamos de instalar.

Agora que já temos o Express baixado, precisamos configurá-lo. Faremos isso criando um módulo do Node.js com essa finalidade.

2.4 Criando o módulo de configuração do Express

Vamos criar o arquivo config/express.js que conterá o código do nosso módulo de configuração do Express:

```
// config/express.js
var express = require('express');
```

O código anterior nos dá uma referência ao módulo do Express através da função **require**. A função require tem como tarefa carregar os módulos de que precisamos em nosso script.

> **O módulo CommonJS**
>
> O Node.js utiliza o padrão *CommonJS* (http://wiki.commonjs.org/wiki/Modules/1.1) para criação de módulos.

A variável express armazena uma função que, ao ser chamada, retorna uma instância do Express:

```
// config/express.js
var express = require('express');
var app = express();
```

Até agora só indicamos que nosso módulo depende do Express, porém, ainda não programamos qualquer funcionalidade.

A responsabilidade do módulo será retornar uma instância configurada do Express. Desta maneira, teremos as configurações centralizadas em um único arquivo:

```
// config/express.js
var express = require('express');

module.exports = function() {
    var app = express();
    return app;
};
```

Repare que a linha que instancia o Express está dentro de uma função passada como parâmetro para o objeto module.exports, disponível implicitamente em cada módulo. Tudo que for adicionado em sua propriedade exports estará visível fora do módulo; em nosso caso, o que estaremos disponibilizando é uma função que ao ser chamada retornará uma instância do Express. E agora? Temos um módulo, mas onde utilizá-lo?

Criando server.js

Vamos criar na pasta raiz do projeto o arquivo server.js. É neste arquivo que subiremos o servidor através do Node.js associando-o à instância do Express:

```
// server.js
var http = require('http');
var app = require('./config/express')();

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

Declaramos duas dependências no arquivo. A primeira é do módulo http responsável por subir nosso servidor, e a segunda, o módulo que acabamos de criar, com a diferença de que passamos o caminho de nosso arquivo express.js omitindo a extensão; inclusive, por seu retorno ser uma função, usamos () para executá-la.

A função http.createServer recebe como parâmetro um request listener. Em nosso caso, é a instância do Express que será aplicada ao evento request disparada em cada requisição.

Agora podemos rodar nossa aplicação através do comando node seguido do nome do script server.js. Você pode omitir a extensão se preferir:

```
node server
```

No terminal, será exibida a mensagem:

```
Express Server escutando na porta undefined
```

> **PZ Z o u u o**
>
> Você pode parar o servidor no terminal pressionando CONTROL + C ou COMMAND + C no MAC OSX.

Repare que o servidor não encontrou a configuração da porta, por isso o valor undefined. Não é apenas a configuração da porta que está faltando, falta adicionar as configurações mínimas para o funcionamento do Express.

2.5 E u ou o Z f u u u oo u - Z u

Podemos agrupar as configurações do Express em três grupos: **variáveis de ambiente**, **middlewares** e **rotas**, este último veremos mais à frente.

Variáveis de ambiente

Variáveis de ambiente são usadas para inicializar configurações padrões do Express. Vamos configurar a porta da aplicação que estava faltando no código anterior:

```
// config/express.js
var express = require('express');

module.exports = function() {
    var app = express();
```

```
    // variável de ambiente
    app.set('port', 3000);
};
```

Adicionamos ou modificamos variáveis de ambiente através da função app.set, que recebe uma chave e um valor. A chave é a variável de ambiente que queremos modificar, o valor, o que desejamos atribuir à chave.

No exemplo, configuramos o Express para escutar na porta 3000. Mas do que adianta um servidor escutando na porta 3000 se o usuário não consegue acessar conteúdo? É aí que entra nosso primeiro middleware.

O middleware express.static

Precisamos fazer com que os arquivos dentro da pasta public sejam acessíveis pelo usuário através do navegador. Realizamos esta configuração através da função app.use, que recebe como parâmetro o middleware express.static:

```
// config/express.js
var express = require('express');

module.exports = function() {
  var app = express();

  // configuração de ambiente
  app.set('port', 3000);

  // middleware
  app.use(express.static('./public'));

  return app;
};
```

Repare que o middleware express.static recebe como parâmetro nossa pasta ./public. O ./ referencia a pasta onde nosso script foi executado pelo comando node, em nosso caso, a pasta contatooh.

Agora que já temos nosso servidor configurado para compartilhar recursos estáticos, vamos criar a página **public/index.html** com uma mensagem de boas-vindas:

```
<!-- public/index.html -->
<!doctype html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Bem-vindo</title>
</head>
<body>
    <h1>Bem-vindo ao Contatooh</h1>
</body>
</html>
```

Podemos subir o servidor mais uma vez:

```
node server
```

Agora, ele exibirá corretamente a mensagem:

```
Express Server escutando na porta 3000
```

Basta acessar a URL:

```
http://localhost:3000
```

Pronto! Já temos nosso servidor Express rodando e compartilhando páginas estáticas, mas nem sempre isso é suficiente. Muitas vezes precisamos de páginas dinâmicas construídas a partir de algum dado. É o que veremos a seguir.

2.6 View e template engine

Express suporta uma grande variedade de templates engines. Utilizaremos o engine **EJS** (http://embeddedjs.com) que possui uma sintaxe similar ao HTML, facilitando sua aceitação entre designers. Precisamos instalá-lo na pasta raiz do nosso projeto através do npm:

```
npm install ejs@0.8 --save
```

No Express, template engines são configurados em variáveis de ambiente. Elas são adicionadas ou alteradas através da função app.set:

```
// config/express.js

// abaixo do middleware express.static
app.set('view engine', 'ejs');
app.set('views','./app/views');
...
```

No código, primeiro dizemos que a view engine utilizada é ejs. Em seguida, de□nimos o diretório onde □carão nossas views, em nosso casso, ./app/views. Tudo através de variáveis de ambiente.

No lugar de escrevermos outro HTML, moveremos o arquivo public/index.html para a pasta app/views renomeando sua extensão para .ejs:

```
<!-- app/views/index.ejs -->
<!doctype html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Bem-vindo</title>
</head>
<body>
    <h1>Bem-vindo ao Contatooh</h1>
</body>
</html>
```

Um template normalmente possui lacunas que precisam ser preenchidas, é como se fosse um contrato, memorando ou carta na qual apenas algumas informações são alteradas, mantendo-se grande parte do documento original. Vamos indicar que nossa view depende do nome do projeto usando a sintaxe <%=nome %>:

```
<h1>Bem-vindo ao <%=nome %></h1>
```

Pronto, mas temos um problema. Antes, conseguíamos acessar a página index.html porque ela estava dentro da pasta pública; agora, ninguém poderá acessar nossa view, porque ela está em uma pasta diferente. Precisamos tornar público um caminho de acesso para nossa página e fazemos isso no Express através de **rotas**.

2.7 C□□□□□□Z□o□ □□□ Z□

Precisamos con□gurar uma rota de acesso para nossa view index.ejs. Poderíamos fazer isso dentro de express.js, já que temos uma instância do Express con□gurada, mas vamos utilizar um módulo separado, chamado app/routes/home.js. Como as con□gurações de rotas são aplicadas diretamente na instância do Express, nosso módulo precisa receber essa instância como parâmetro:

```
// app/routes/home.js
module.exports = function(app) {

}
```

> **E□u □□ 3.X u □□□ Z□**
>
> A versão 3.X do Express não vem com seu middleware para controle de rotas ativado por padrão. Sua ativação é simples:
>
> ```
> // config/express.js
> ...
> // necessário apenas na versão 3.X
> app.use(app.router);
> ...
> ```

Agora, precisamos importar o módulo dentro de config/express.js passando como parâmetro nossa instância con□gurada do Express:

```
// config/express.js
...
var home = require('./app/routes/home')
...
// abaixo da configuração do último middleware
home(app);
...
```

Já temos tudo encaixado, vamos configurar a rota através da função **app.get**. Em nosso caso, queremos responder à URL /:

```
// app/routes/home.js
module.exports = function(app) {
  app.get('/')
}
```

A função app.get equivale ao verbo GET do http, isto é, uma função preparada para lidar com requisições do tipo GET. Veremos outros tipos à medida que precisarmos.

Neste exemplo, quando o usuário digitar a URL http://localhost:3000/ o Express verificará se há alguma rota registrada como / e executará uma ação associada à rota. Mas espere! Não indicamos qual código será executado. No modelo MVC, este é o papel do *controller*.

2.8 Criação do controller

No modelo MVC, o controller é a ponte de ligação entre nossa página (view) e nossos dados (model). Criaremos nosso controller no arquivo app/controllers/home.js. Repare que ele tem o mesmo nome do arquivo de rotas, justamente para indicar uma ligação entre ambos, mas poderíamos ter escolhido outro nome. Nosso arquivo será um módulo, com a diferença de que não receberá a instância do Express como parâmetro.

```
// app/controllers/home.js

module.exports = function() {
  var controller = {};
  return controller;
}
```

O módulo retorna um objeto. É nele que adicionamos uma ou mais ações que podem ser chamadas por uma rota acessada pelo usuário. Vamos aplicar uma ação que será responsável pelo retorno da nossa página index.ejs até agora inacessível para o mundo externo:

```
module.exports = function() {
  var controller = {};
  controller.index = function(req, res) {
    // retorna a página index.ejs
  };
  return controller;
}
```

Adicionamos no objeto controller a propriedade **index**, que armazena uma função com dois parâmetros: o primeiro, o objeto que representa o fluxo da **requisição**; o segundo, o de **resposta**. É através do objeto res que enviaremos a página index.ejs:

```
module.exports = function() {
  var controller = {};
  controller.index = function(req, res) {
    res.render('index', {nome: 'Express'});
  };
  return controller;
}
```

Lembre-se que não podemos devolver diretamente a página index.ejs, porque ela não está completa e precisa do dado nome. É por isso que usamos a função res.render. Ela recebe dois parâmetros: o primeiro é o **nome da view** que será retornada e o segundo, um **objeto JavaScript com dados** que será consumido por esta view.

Uma dúvida que pode aparecer agora é como o Express encontrará o arquivo de nossa view. Por padrão, ele adiciona o sufixo .ejs (nosso template engine) ao nome do arquivo, procurando-o dentro da pasta app/views. Lembra da variável de ambiente views?

Processando o template antes do envio

Antes da resposta ser enviada, a view engine processará o template e preencherá a expressão <%=nome %>procurando no objeto passado como parâmetro para a função res.send a propriedade nome. No final, o que será devolvido para o navegador como resposta é um novo HTML com a expressão <%= nome %>substituída por Express.

Por fim, exibindo o código-fonte no navegador:

```
<!doctype html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Bem-vindo</title>
</head>
<body>
    <h1>Bem-vindo ao Express</h1>
</body>
</html>
```

Dica

Precisamos reiniciar o servidor toda vez que realizarmos alterações. Podemos resolver este problema usando o **nodemon**:

```
npm install nodemon -g
nodemon server.js
```

O único detalhe é que somos obrigados a informar a extensão do arquivo.

Tudo é Controller

Repare que nosso template é montado primeiro no servidor e depois enviado para o navegador. Veremos algo diferente quando trabalhamos com AngularJS. Nele, a composição do template é feita no navegador, não no servidor.

Você já deve ter reparado que, toda vez que criarmos um arquivo de rota em separado, ele precisará importar o módulo do controller que será utilizado. Ainda não temos nenhum model criado; se tivéssemos, ele também precisaria ser importado, mas dentro do arquivo do controlller:

```
npm install express-load@1.1 --save
```

Em seguida, precisamos importar o módulo dentro do nosso arquivo de configuração do Express:

```
// config/express.js
var load = require('express-load');
```

Por fim, substituímos a chamada à função home(app) pela função **load**:

```
...
load('models', {cwd: 'app'})
  .then('controllers')
  .then('routes')
  .into(app);

return app;
```

A função load carregará todos os scripts dentro das pastas app/models, app/controllers e app/routes. No final, a função into adiciona dinamicamente na instância do Express propriedades que apontam para esses módulos.

Ordem de carregamento

Um ponto importante é que precisamos carregar as pastas seguindo a ordem *models, controllers e routes*, caso contrário não conseguiremos, por exemplo, ter acesso aos nossos controllers em nossas rotas caso os módulos com nossos controllers tenham sido carregados por último.

Pasta padrão

O parâmetro {cwd: 'app'} foi necessário para mudar o diretório padrão, pois a função procura as pastas no diretório raiz contatooh e precisamos que ela considere a pasta contatooh/app.

Modificando nosso arquivo de rotas

Precisamos alterar nosso arquivo app/routes/home.js, que não fará mais uso da função require para carregar seu controller. Em vez disso, ela procurará o controller diretamente na instância do Express passada como parâmetro:

```
// app/routes/home.js
module.exports = function(app) {
  var controller = app.controllers.home;
  app.get('/', controller.index);
};
```

Podemos testar o projeto para verificar se tudo continua funcionando:

http://localhost:3000/

Até aqui entendemos como o Express funciona e como configurá-lo, inclusive organizamos nossa aplicação em uma estrutura que facilitará sua manutenção. Agora precisamos criar as rotas de que nossa aplicação precisa. É o que veremos na próxima seção.

2.10 Listagem de contatos

Nosso servidor precisa retornar uma lista de contatos que mais tarde será consumida pelo nosso front-end usando AngularJs. Disponibilizaremos este recurso através de um identificador. No protocolo HTTP, usamos URLs para isso:

http://localhost:3000/contatos

Lembre-se que no Express esta URL é chamada de **rota** (route) e precisa ser configurada. Vamos criar o arquivo app/routes/contato.js:

```
// app/routes/contato.js
module.exports = function(app) {
  var controller = app.controllers.contato;
  app.get('/contatos', controller.listaContatos);
};
```

Z□□.□□u

Como vimos, app.use é usado para associar middlewares à nossa instância do Express. Ele também pode ser usado para ativar middlewares para uma rota especí□ca. A diferença é que ele considerará a rota independente do verbo utilizado, seja ele GET, PUT, DELETEou POST.

```
app.use('/'   , function(req, res, next) {
        console.log('middleware A');
        next();
 });
 app.use('/',      function(req, res, next) {
        console.log('middleware B');
        next();
 });
```

Este código exibirá no console as mensagens:

```
middleware A
middleware B
```

Repare o parâmetro **next**. É uma função que ao ser executada chama o próximo middleware da pilha para processar a requisição. Se comentarmos a chamada à next() no primeiro middleware, o console apenas exibirá o texto 'middleware A', deixando de chamar o próximo da pilha.

Lembre-se que não precisamos importar o módulo dentro de config/express, já que o express-load se encarregará de carregar o novo script, tornando-o disponível na instância do Express como app.controlllers.contato.

O problema é que o código anterior referencia um controller que ainda não existe. Precisamos criá-lo, através do arquivo app/controllers/contato.js:

```
// app/controllers/contato.js
module.exports = function() {
  var controlller = {};
```

Nosso controller está quase pronto, falta retornar a lista de contatos.

res.json

Diferente do controller da home que devolve uma página HTML dinamicamente gerada pelo servidor, desta vez retornaremos uma lista de dados no formato JSON, que mais tarde será consumida no lado do cliente através do AngularJS.

O Express possui a função res.json especializada no envio deste tipo de dado:

```
...
module.exports = function() {
  var controlller = {};
  controller.listaContatos = function(req, res) {
      res.json(contatos);
  };
  return controller;
};
```

Acessando o endereço (http://localhost:3000/contatos) em nosso navegador, nossa lista é exibida no seguinte formato:

```
[
  {"_id":1,"nome":"Contato Exemplo 1",
   "email":"cont1@empresa.com.br"},
  {"_id":2,"nome":"Contato Exemplo 2",
   "email":"cont2@empresa.com.br"},
  {"_id":3,"nome":"Contato Exemplo 3",
   "email":"cont3@empresa.com.br"}
]
```

□u □.□□□□ □. □u □.□u □o

De acordo com a documentação do Express, res.json é idêntico a res.send quando um objeto ou array é passado. A diferença é que o primeiro explicitamente converte tipos não-objeto como null e undefined. Usamos res.json também para deixar clara nossa intenção de trabalhar com este tipo de dado.

Precisamos criar agora um recurso que permita retornar apenas um contato dado seu identi□cador. Alguma sugestão de nome? Deve ter passado em sua cabeça vários nomes, mas podemos adotar um padrão que ajudará a uniformizar nossas regras de de□nição de recurso. Este padrão se chama **REST**.

2.11 U□□□□h□ □□f□u REST API

O acrônimo REST (*Representational State Transfer*) foi criado por Roy Fielding, um dos principais criadores do protocolo HTTP. Ele de□ne um conjunto de operações aplicáveis a todos os recursos de informação utilizando o protocolo HTTP. As operações mais utilizadas são DELETE, GET, POSTe PUT. Em REST, cada recurso deve ter um identi□cador único, o que em HTTP é sua URL de acesso.

Trazendo essa de□nição do REST para nosso servidor Express, temos além da função app.get as funções **app.delete**, **app.put** e **app.post**, cada uma correspondendo a um verbo HTTP utilizado pelo padrão REST, inclusive somos capazes de criar URLs únicas em nossas rotas, que nada mais são do que identi□cadores únicos.

PZ□Z □Zfu □□Z□□

Você pode querer se aprofundar na API REST, que vai muito além da de□nição de nome de recursos. Você pode consultar o livro **REST: Construa API's inteligentes de maneira simples** também disponível na Casa do Código.

Já temos configurada a rota /contatos, mas precisamos criar outra para acessar um contato individualmente. E se eu te dissesse que podemos usar a "mesma" URL /contatos. Confuso? Não se preocupe, é o que veremos a seguir.

2.12 Rotornando contato da lista

Imagine que para obter um contato você tenha que passar pela lista de contatos, com a diferença de que está interessado apenas no contato com determinado identificador. Sendo assim, uma URL condizente com essa ideia seria /contatos/1. Se quiséssemos o contato com identificador 30, acessaríamos a URL /contatos/30.

Essa maneira é perfeitamente aceita dentro do padrão REST, o problema é que teríamos que configurar uma rota para cada contato cadastrado em nosso sistema, algo impraticável.

Para solucionar este problema, o Express permite registrar rotas seguindo este padrão utilizando um curinga no lugar do identificador do recurso como /contatos/:id. Adicionando a rota em app/routes/contato.js:

```
// app/routes/contato.js
...
module.exports = function(app) {
  var controller = app.controllers.contato;

  app.get('/contatos', controller.listaContatos);
  app.get('/contatos/:id', controller.obtemContato);
};
```

Isto significa que URLs como (http://localhost:3000/contatos/2) não dispararão a rota /contatos, mas aquela com o curinga. Mais que isso, ainda será possível capturar no controlador o parâmetro passado através de req.params.id. Neste caso, id é o nome do curinga utilizado na rota. Adicionando a nova ação em app/controllers/contato.js:

```
// app/controllers/contato.js
  ...
  controller.obtemContato = function(req, res) {
```

```
    console.log(req.params.id);
};
...
```

Agora, só precisamos criar nossa lógica que devolve o contato com o id enviado como parâmetro. Uma implementação possível é a seguinte:

```
controller.obtemContato = function(req, res) {
  var idContato = req.params.id;
  var contato = contatos.filter(function(contato) {
    return contato._id == idContato;
  })[0];
  contato ?
    res.json(contato) :
    res.status(404).send('Contato não encontrado');
};
```

Nesse código, quando um contato não é encontrado, alteramos o status da resposta para 404 (não encontrado) através da função `res.status` e enviamos como resposta uma mensagem com uma pista do problema ocorrido.

Já podemos testar:

```
http://localhost:3000/contatos/2
```

Será exibido no navegador o JSON:

```
{"_id":2,"nome":"Contato Exemplo 2",
 "email":"cont2@empresa.com.br"}
```

Consideração sobre os verbos DELETE e PUT

Nem todos os navegadores suportam os verbos `DELETE` e `PUT`, inclusive, há redes que filtram requisições destes tipos. Uma solução empregada é usar o verbo `POST`, mas adicionando no *header* da requisição o `Content-type` **X-HTTP-Method-Override**. Nele, informamos qual verbo pretendemos utilizar, por exemplo, `DELETE`. O servidor precisa estar preparado para extrair esta informação e chamar a rota para o tipo de verbo desejado.

No Express, usamos o middleware **method-override** (http://https://github.com/expressjs/method-override) que cuidará automaticamente deste processo para nós, garantindo que as rotas para os verbos sejam chamadas corretamente. Para funcionar, ele depende de que a requisição seja parseada através do middleware **body-parser** (https://github.com/expressjs/body-parser) . Este último popula req.body com os parâmetros do POST.

Ambos são instalados através do npm executando-o dentro da pasta raiz do projeto:

```
npm install body-parser@1.6 method-override@2.1 --save
```

Por fim, precisamos ativar esses middlewares na configuração do Express:

```
// config/express.js
...
var bodyParser = require('body-parser');
...
  // middleware
  app.use(express.static('./public'));
  app.use(bodyParser.urlencoded({extended: true}));
  app.use(bodyParser.json());
  app.use(require('method-override')())
...
```

Repare no código anterior que estamos solicitando ao bodyParser que realize o parse de json e requisições com o corpo x-ww-form-urlencoded. Isso permite acessar os dados da requisição através de req.body, algo que veremos na prática mais tarde.

[illegible]

As primeiras versões do body-parser para Express 4 adotavam como padrão da chave extended o valor true que utiliza como parser a biblioteca qs. Porém, quando false, utiliza a biblioteca querystring. As versões mais recentes do middleware imprimirão no console uma mensagem de aviso solicitando a escolha da biblioteca, algo que podemos resolver passando para a chave true ou false. Outros middlewares do Express também estão deixando de assumir valores padrões, inicialmente exibindo apenas um aviso de que no futuro o valor padrão não será mais suportado.

[illegible] 3.x [illegible]

Até a versão 3.X, o Express herdava body-parser e method-override do Connect e eram ativados da seguinte maneira:

```
app.use(express.methodOverride());
app.use(express.bodyParser());
```

Repare que não era necessário indicar o parser de json nem urlencoded. O padrão era para ambos, porém muitos desenvolvedores achavam que ele suportava mais coisas com multipart, o que não era verdade.

Agora que nosso servidor lista e retorna contatos individualmente, podemos dar uma pausa no server-side e focar nossa atenção no client-side. O primeiro passo será obter nossas bibliotecas front-end.

CZ£ –¶ 3

Bower: gerenciador de dependências front-end

"O homem realmente culto não se envergonha de fazer perguntas também aos menos instruídos."
– Lao-tsé

Quantas vezes você já precisou abrir seu navegador para procurar alguma biblioteca front-end para depois baixá-la? Quanto tempo gastou para achar o link de download no site? Era a versão que você queria? Salvou o arquivo no diretório correto no seu projeto? E quando precisou atualizar a biblioteca? Repetiu o mesmo ciclo? Fez a mesma coisa com todas as outras bibliotecas da sua aplicação? Resolveu con□itos? E se eu lhe dissesse que não precisa ser assim? Foi pensando em questões como essas que a equipe do Twitter

(https://twitter.com) criou um gerenciador de pacotes voltado para front-end chamado **Bower**.

O Bower (http://bower.io) é um gerenciador de pacotes para web voltado para front-end que realiza grande parte das tarefas que faríamos manualmente, inclusive gerencia as dependências de sua aplicação. Outro ponto interessante é que ele não gerencia pacotes exclusivamente JavaScript, mas também CSS e HTML.

B[illegible]

Podemos comparar o Bower com npm, com a diferença de que o primeiro é voltado para pacotes front-end e o segundo para pacotes back-end, apesar de alguns desenvolvedores tentarem utilizar o npm para as duas finalidades.

Faz todo sentido incluirmos o Bower dentro da MEAN Stack já que ele nada mais é do que um módulo do Node.js. Veremos a seguir como configurá-lo.

3.1 Instalando o Bower

Sendo um módulo do Node.js, a instalação do Bower é feita através do npm:

```
npm install bower@1.3 -g
```

O parâmetro -g torna o módulo acessível em qualquer local em nosso terminal, isto é, globalmente. Isso faz todo sentido, já que queremos baixar nossas dependências a qualquer momento em qualquer um de nossos projetos.

[illegible] -g

Para que o parâmetro -g funcione, você precisa ter permissão de administrador, caso contrário a instalação falhará.

Podemos testar a instalação do Bower através do comando bower:

```
$ bower

Usage:

    bower <command> [<args>] [<options>]

Commands:
    (exibe uma lista com todos os comandos possíveis)
```

O comando anterior exibirá uma lista com vários parâmetros que podem ser utilizados. Focaremos nos mais utilizados, mas primeiro, precisamos inicializar o Bower em nosso projeto. É o que veremos a seguir.

3.2 f□ □□□ u □□□Z□ ou □u □ox□hZ□

Poderíamos já começar a utilizar o Bower para baixar nossas dependências, mas queremos que todas □quem listadas em um arquivo, algo semelhante ao que □zemos com o arquivo package.json, que guarda o nome e a versão dos módulos utilizados pelo nosso servidor.

No Bower, o arquivo que lista nossas dependências é chamado bower.json. Dentro da pasta raiz do projeto, ele pode ser criado pelo comando bower init:

```
$ bower init
[?] name: contatooh
[?] version: 0.0.0
[?] description:
[?] main file:
[?] what types of modules does this package expose?
[?] keywords:
[?] authors: flaviohenriquealmeida
[?] license: MIT
[?] homepage:
[?] set currently installed components as dependencies? Yes
[?] add commonly ignored files to ignore list? Yes
[?] would you like to mark this package as private which
```

```
   prevents it from being
[?] would you like to mark this package as private which
   prevents it from being
accidentally published to the registry? Yes

{
   name: 'contatooh',
   version: '0.0.0',
   authors: [
      'flaviohenriquealmeida <flaviohenriquealmeida@gmail.com>'
   ],
   license: 'MIT',
   private: true,
   ignore: [
      '**/.*',
      'node_modules',
      'bower_components',
      'test',
      'tests'
   ]
}

[?] Looks good? (Y/n)
```

O arquivo fará uma série de perguntas como o nome do projeto e a sua versão. Podemos dar ENTERpara todas as perguntas, mas vale o esclarecimento de algumas delas:

- **What types of modules does this package expose?** (Qual tipo de módulo este pacote expõe?): com o Bower, também podemos criar pacotes como módulos que podem ser consumidos por outras aplicações. No universo JavaScript, existem alguns formatos para criação de módulos. O Bower deseja saber qual formato seu pacote utilizará para dar uma pista para aqueles que o forem utilizar. Como nossa aplicação não será um pacote, podemos dar ENTERpara esta pergunta.

- **Set currently installed components as dependencies?** (Adicionar os componentes já instalados como dependências?): pode ser que você já

tenha baixado pacotes antes de criar o bower.json e o Bower fornece a opção de incluir no arquivo os pacotes já baixados.

- **Would you like to mark this package as private which prevents it from being accidentally published to the registry?** (Você gostaria de marcar este pacote como privado, o que evitará que ele seja acidentalmente publicado no registro?): esta é mais uma opção que só faz sentido caso nosso código fosse empacotado. Ela evita que pacotes privados sejam publicados, logo, acessíveis por qualquer um através do Bower.

> **f [illegible]**
>
> O Bower utilizava nas versões anteriores à 0.9.0 o arquivo component.js para listar as dependências da sua aplicação. A partir desta versão, component.js foi renomeado para bower.json.

Agora que temos nosso arquivo bower.json criado, já podemos baixar nossas dependências front-end.

3.3 [illegible]

Temos o Bower instalado e o arquivo bower.json dentro da pasta contatooh. Nosso projeto utilizará duas bibliotecas: **AngularJS** e **Bootstrap**. A primeira é uma biblioteca JavaScript; a segunda, uma biblioteca CSS. Vamos instalar o AngularJS primeiro e mais tarde o Bootstrap.

Utilizamos o comando bower install passando como parâmetro o **nome** da biblioteca seguido opcionalmente de # e o número da **versão**. Utilizaremos o parâmetro --save que adicionará a biblioteca como dependência em bower.json. Queremos a versão 1.3 do AngularJS:

```
$ bower install angular#1.3 --save

bower not-cached git://github.com/angular/bower-angular.git#1.3
bower resolve git://github.com/angular/bower-angular.git#1.3
bower download https://github.com/angular/bower-angular/
```

archive/v1.3.2.tar.gz
bower extract angular#1.3 archive.tar.gz
bower resolved git://github.com/angular/
bower-angular.git#1.3.2

Visualizando bower.json, vemos que o AngularJS foi adicionado como dependência:

```
{
  ...
  ...
  ...
  "dependencies": {
    "angular": "1.3"
  }
}
```

Vamos analisar o resultado do comando anterior. O Bower criou por convenção a pasta bower_components dentro do diretório onde foi chamado, em nosso caso, contatooh, a raiz de nosso projeto. Dentro dela, ele criou a pasta angular que contém scripts minificados e não minificados. O problema é que, por ser uma biblioteca front-end, ela precisa estar dentro da pasta public. Precisamos alterar a pasta padrão criada pelo Bower criando um novo arquivo de configuração, o que veremos a seguir.

3.4 Alterando a pasta destino com .bowerrc

Podemos alterar o caminho e o nome da pasta criada pelo Bower para armazenar nossas dependências através do arquivo oculto .bowerrc. Vamos criar o arquivo dentro da raiz do projeto com a seguinte estrutura:

```
// contatooh/.bowerrc
{
    "directory": "public/vendor"
}
```

Repare que .bowerrc também é um JSON assim como bower.jon com a diferença de que o primeiro possui apenas uma propriedade chamada

directory. É nesta propriedade que de□nimos o local onde serão baixados os pacotes do projeto. Em nosso caso, além de alterarmos o caminho da pasta, também mudamos o nome do diretório para public/vendor.

Precisamos apagar a pasta bower_components para, em seguida, baixar novamente nossas dependências fazendo com que todas □quem na pasta que acabamos de con□gurar. Isso é feito através do comando bower install:

```
$ bower install

bower cached git://github.com/angular/bower-angular.git#1.3.2
bower validate 1.3.2 against git://github.com/angular/
                                                bower-angular.git#1.3
bower install angular#1.3.2
angular#1.3.2 public/vendor/angular
```

Assim como npm install, que lê as dependências no lado do servidor no arquivo package.json, o Bower baixará todas as dependências listadas em bower.json, só que desta vez as salvando dentro da pasta public/vendor.

O Bower possui outros comandos que podem ser úteis e vale a pena dar uma olhada em alguns deles.

3.5 O□□ □□□ h□□Z□o□□

Vimos que o comando bower install pacote#versao --save se encarrega de baixar nossos pacotes, adicionando-os em bower.json. Há outros comandos que podem ser úteis no work□ow do programador front-end. Vejamos mais alguns deles.

search

Muitas vezes não sabemos ao certo o nome de uma biblioteca e, nesses casos, teríamos que pesquisar na web. Bower possui o comando search que lista todas as bibliotecas registradas que possuem o texto procurado, por exemplo:

```
bower search angular
```

Serão listados no terminal bibliotecas que contenham como parte de seu nome a palavra angular.

```
$ bower search angular
...
angularjs-google-maps git://github.com/jvandemo/
                                                    angularjs-google-maps
dt-angular-ui git://github.com/dt-bower/dt-angular-ui.git
angular-aop git://github.com/mgechev/angular-aop
angular-storage git://github.com/Aspera/angular-storage.git
...
```

info

Quais versões de uma determinada biblioteca estão disponíveis pelo Bower? Podemos ter essa informação facilmente pelo comando:

```
bower info angular
```

Esse comando listará no terminal todas as versões disponíveis e que podem ser baixadas através do Bower:

```
$ bower info angular

bower cached      git://github.com/angular/bower-angular.git#1.3.2
bower validate   1.3.2 against git://github.com/angular/
                                                    bower-angular.git#*

{
  name: 'angular',
  version: '1.3.2',
  main: './angular.js',
  ignore: [],
  dependencies: {},
  homepage: 'https://github.com/angular/bower-angular'
}

Available versions:
  - 1.3.3-build.3574+sha.e3764e3
  - 1.3.3-build.3573+sha.4d4e603
```

```
  - 1.3.3-build.3572+sha.4d885cb
(outras versões)
```

uninstall

O que fazer quando uma biblioteca deixar de ser uma dependência do nosso projeto? Precisamos apagar sua pasta com todos os seus arquivos, inclusive alterar nosso arquivo bower.json removendo a biblioteca.

Podemos fazer tudo numa tacada só com o comando:

```
bower uninstall angular --save
```

```
$ bower uninstall angular --save
bower uninstall         angular
```

--offline

Todo pacote baixado pelo Bower fica armazenado em um cache que pode ser utilizado offline, muito bem-vindo quando estamos sem acesso à internet e não queremos perder tempo. Para isso, basta adicionar o parâmetro --offline em conjunto com o comando bower install:

```
bower install angular#1.3 --save --offline
```

```
$ bower install angular#1.3 --save --offline

bower cached              git://github.com/angular/
                                                bower-angular.git#1.3.2
bower install             angular#1.3.2
angular#1.3.2 public/vendor/angular
```

Pronto, já temos tudo de que precisamos parar começar nossa jornada pelo AngularJS.

CZ£ –¶ 4

AngularJS: o framework MVC da Google

"A verdade não resulta do número dos que nela creem."
– Galileu

É quase certo de que durante sua trajetória como desenvolvedor web você já tenha trabalhado com JavaScript no navegador, ainda mais sendo a língua franca da web. Mesmo com o advento do Node.js, o código que escrevemos no lado do cliente manipula algo que não existe no lado do servidor: o *Document Object Model* (DOM). Vamos recapitular este velho conhecido e a maneira tradicional como manipulá-lo.

4.1 Um velho conhecido: DOM

O DOM é uma árvore de elementos, isto é, um espelho em memória de nossa página criado automaticamente pelo navegador quando ela é carregada. Inclusive, ele possui funções e propriedades que, ao serem modificadas, disparam alterações instantâneas no que é exibido para o usuário.

Vejamos um exemplo de manipulação do DOM que incrementa o total de contatos cadastrados para cada clique no botão.

Primeiro o HTML:

```
<!-- página html -->
<button class="botao-grava">Novo</button>
<p class="contatos">Contatos cadastrados: 0 </p>
```

Agora o JavaScript:

```
var contatos = document.querySelector('.contatos');
var total = 0;
var botao = document.querySelector('.botao-grava');
botao.addEventListener('click', function(event) {
  total++;
  contatos.textContent = 'Contatos cadastrados: ' + total;
});
```

Apesar de ser JavaScript puro, o código é totalmente funcional, mas há aqueles que utilizam bibliotecas específicas para manipulação de DOM. Uma muito famosa é o jQuery.

jQuery

Historicamente, o DOM não teve um bom começo, com cada navegador implementando funções e propriedades a seu bel-prazer, o que acabou gerando incompatibilidades entre eles. Este foi um dos motivos que levaram à criação do jQuery (http://jquery.com) e de outras **bibliotecas de manipulação de DOM** que visam blindar o programador das idiossincrasias da API DOM entre navegadores, inclusive adicionando açúcares sintáticos.

O exemplo anterior agora com jQuery:

```
var contatos = $('.contatos');
contatos.data('total', 0);
$('.botao-grava').click(function() {
  var total = contatos.data('total') + 1;
  contatos.data('total', total);
  contatos.text('Contatos cadastrados: ' + total);
});
```

Por fim, mais um exemplo de manipulação de DOM usando apenas jQuery desta vez. O código deve exibir o resultado da multiplicação entre dois números informados pelo usuário. Primeiro o HTML:

```
<p>
  Quando é<input class="numero"> vezes <input class= "numero">
</p>
<p>Resultado: <span class="resultado" ></span></p>
```

Agora, o JavaScript com jQuery:

```
var numerosEl = $('.numero');
$(numerosEl).on('keyup', function() {
    var resultado = 1;
    $.each(numerosEl, function() {
      resultado*= $(this).val();
    });
    $('.resultado').text(resultado);
});
```

Mais uma vez, temos um código funcional, apesar disso, nem tudo é perfeito na manipulação do DOM.

4.2 Dificuldades na manipulação do DOM

Apesar de ser uma tarefa rotineira para programadores front-end, a manipulação do DOM traz algumas dificuldades.

Presos à estrutura

Nos exemplos anteriores, por mais que tenhamos utilizado classes nos seletores para não ficarmos amarrados à estrutura do documento, ainda somos obrigados a conhecê-lo para saber que o elemento X possui a classe tal e, portanto, pode ser selecionado:

```
// preciso olhar a estrutura e ver que o elemento tem a classe
// .numero
var numerosEl = $('.numero');
```

Inclusive precisamos saber qual função chamar para obter o valor do elemento, por exemplo, `val()` ou `text()` usando jQuery:

```
// se fosse um input, seria val()
$('.resultado').text(resultado);
```

E se quisermos ainda testar nosso código?

Testabilidade

Outro ponto que salta à vista é que a não separação entre a lógica e os elementos do DOM torna o teste do código uma tarefa não trivial:

```
// obtém um elemento do DOM
var numerosEl = $('.numero');
```

Já pensou em como realizar um teste de unidade no código que vimos? Como simular a página durante os testes? E por aí vai.

Consistência entre model e view

Também somos os responsáveis em garantir a consistência entre o dado e sua apresentação. Em outras palavras, toda vez que o model for atualizado, precisaremos atualizar sua apresentação na view.

```
var numerosEl = $('.numero');

    var resultado = 1;
```

```
    // alterando o equivalente ao model
    $.each(numerosEl, function() {
      resultado*=$(this).val();
    });

    // sincronizando o model com a view
    $('.resultado').text(resultado);
});
```

Se pensarmos no modelo MVC, precisamos garantir a consistência entre o model e a view.

Ausência de um padrão

Outro ponto relevante é que não há nada que imponha um padrão na escrita do código. Por exemplo, a mesma funcionalidade com jQuery que vimos pode ser escrita da seguinte maneira:

```
var numerosEl = $('input[type=text]');
$(numerosEl).keyup(function() {
    var resultado = 1;
    $.each(numerosEl, function() {
      resultado*= $(this).val();
    });
    $('.resultado').text(resultado);
});
```

É claro que a equipe pode definir um, mas terá que transmiti-lo para novos desenvolvedores.

Outra opção: frameworks MVC client-side

Com base nos problemas listados, foram criados frameworks MVC *client-side* como BackBone (http://backbonejs.org) , Amber (http://emberjs.com) , Can (http://canjs.com) entre outros. Um framework que vem ganhando muita força na comunidade é o **AngularJS**. Qual a razão de seu sucesso? Vamos ver!

4.3 CZ□Zh□ u □□□□ □hZ□ o□ A□□□□Z□JS

AngularJS (http://angularjs.org) é um framework MVC client-side que trabalha com tecnologias já estabelecidas: HTML, CSS e JavaScript. Criado na Google e liberado como projeto open-source para o público em 2009, seu foco reside na criação de *Single Page Applications* (SPA), embora isso não o impeça de ser utilizado em outros tipos de aplicação. AngularJS é um *high opinionated framework*, o que signi□ca que ele guia o desenvolvedor de maneira implacável na organização de seu código.

> **S□□□□u PZ□u A□□□□hZ□ □□□ (SPA)**
>
> SPA é uma aplicação entregue ao navegador que não recarrega a página durante seu uso, por isso já carrega todo JavaScript e CSS de que precisa.

Miško Hevery, pai do framework, procurou trazer os mesmos benefícios do modelo MVC aplicados no lado do servidor para o navegador como a **facilidade de manutenção**, **reusabilidade** através de componentes e **testabilidade**, uma vez que a lógica é desacoplada de elementos do DOM. Ainda há recursos como **injeção de dependências**!

Enumerar as características do AngularJS não é tão esclarecedor quanto vê-las na prática, muito menos apresentar todas de uma só vez. Nosso primeiro passo será criar uma nova página com AngularJS habilitado.

4.4 P□u □Z□Z□o□ □ Z□f□u □□ u

Agora que vamos trabalhar com o AngularJS, deixaremos de utilizar views processadas no lado do servidor. Para evitar confusão, vamos renomear os arquivos garantindo que eles não sejam carregados pelo Express e também para que você não os perca:

- app/views/index.ejs para app/views/index.ejs.old
- app/controllers/home.ejs para app/controllers/home.ejs.old

4.5 HZf□□□Z□o□ A□□□Z□JS u□□□□Z □□□Z

Para habilitarmos o AngularJS em nossa página public/index.html, precisamos primeiro importar seu script antes da tag de fechamento </body>:

```
<!-- public/index.html -->
...
<body>
  <h1>Bem-vindo ao Contatooh</h1>

  <button>Novo</button>
  <p>Contatos cadastrados: 0</p>

  <script src="vendor/angular/angular.js"></script>
</body>
...
```

Importar a biblioteca não é su□ciente. Precisamos adicionar o atributo ng-app na área que será controlada pelo AngularJS:

```
<!-- public/index.html -->
...
<html ng-app="contatooh">
  ...
  <body>
    <h1>Bem-vindo ao Contatooh</h1>

    <button>Novo</button>
    <p>Contatos cadastrados: 0</p>

    <script src="vendor/angular/angular.js"></script>
  </body>
</html>
```

Isso é necessário, porque o AngularJS gerencia apenas o bloco com o atributo ng-app. Em nosso exemplo, estamos gerenciando o HTML inteiro, mas poderíamos gerenciar uma área menor, deixando o restante para outros frameworks que também gerenciam o HTML, como Emberjs (http://emberjs.com/) , evitando con□ito. Um ponto curioso é que o atributo

ng-app não existe no HTML, sendo uma **diretiva** do AngularJS. Diretivas podem ser **atributos**, **tags** e até mesmo **comentários especiais** que ampliam o vocabulário do navegador.

Repare que, na diretiva ng-app, atribuímos o valor contatooh, que na verdade é o nome do módulo principal da aplicação, aquele pelo qual realizamos uma série de configurações. Mas onde é que está este módulo? Precisamos criá-lo e é isso que faremos.

4.6 Nosso módulo no AngularJS

Um módulo em AngularJS nada mais é do que um código JavaScript declarado em seu arquivo. Vamos criar o arquivo public/js/main.js importando-o logo em seguida em public/index.html:

```
<!-- public/index.html -->
...
<script src="vendor/angular/angular.js"></script>
<script src="js/main.js"></script>
...
```

Editando o arquivo public/js/main.js vamos programar nosso módulo:

```
// public/js/main.js

angular.module('contatooh', []);
```

Espere um pouco? De onde veio o objeto angular?

O objeto angular

AngularJS disponibiliza o objeto angular globalmente. É através dele que acessamos vários recursos do framework, inclusive o de criação de módulos, justamente aquele de que precisamos.

Criamos um módulo através da função angular.module. Ela recebe dois parâmetros: o primeiro é o **nome do módulo**; o segundo, um **array com todas as suas dependências**. Não temos nenhuma dependência por enquanto, ainda assim, precisamos passar o array **vazio** como parâmetro.

M□o□□□ N□ou .□□□. M □o□□□ A□□□□Z□JS

Não confunda o módulo que criamos no lado do servidor usando Node.js com o módulo do AngularJs. O primeiro roda no server-side, o segundo no client-side.

Repare que o nome do módulo possui o mesmo valor da diretiva ng-app, garantindo a ligação de nossa página com o módulo que acabamos de criar. Nossa próxima tarefa será associar algum dado à nossa view.

4.7 A□□□□Z□u □□u □□□□□ (AE)

Em public/index.html, nosso parágrafo exibe o total de contatos cadastrados, mas o valor está □xo. É necessário que essa informação varie toda vez que o usuário clicar no botão "Novo". Para isso, usaremos uma **Angular Expression (AE)**:

```
<p>Contatos cadastrados: {{total}}</p>
```

Uma AE é representada por {{ }}. Tudo o que estiver entre as chaves duplas será buscado de algum lugar e mesclado com nossa view. Nesse ponto, podemos dizer que nossa view é uma espécie de **template**, já que possui uma lacuna que precisa ser preenchida. Em nosso caso. usamos a AE {{total}}.

Veri□quemos o resultado:

http://localhost:3000

Fig. 4.2: Angular Expression (AE) não avaliada

Você deve ter reparado que a Angular Expression não é exibida no navegador, ela é simplesmente ignorada! Isso é um padrão do AngularJS: **AE não resolvidas ficam em branco**. Ainda precisamos que algum valor seja exibido, mesmo que seja o valor zero inicialmente.

V Z o {{ Z }}u u Z o

Caso você esteja visualizando no navegador o texto {{total}}, é muito provável que haja algum problema na configuração do AngularJS em sua página (importação de script faltando, ng-app apontando para um módulo que não existe etc.).

Se o AngularJS é um framework MVC, quem é o responsável em disponibilizar os dados para a view? O controller! Precisamos de um controller que disponibilize o dado de que nossa AE precisa.

4.8 N u h u

A diretiva ng-app apenas indica que a tag html é gerenciada pelo AngularJS, inclusive todos seus elementos filhos, mas não indica **quem fornecerá os dados** ou **quem responderá às ações** levadas com esses elementos. Em AngularJS, dizemos que estas são responsabilidades de um **controller**.

A diretiva ng-controller

Associamos um controller a um elemento de nossa página através da diretiva ng-controller. Em nosso caso, ela será associada à tag <body>:

```
<! -- public/index.html -->
...
<body ng-controller="ContatosController">
...
```

No exemplo anterior, o controller **ContatosController** será o responsável em fornecer o dado {{total}} para view, sendo assim, precisamos criá-lo. É uma boa prática do AngularJS criar cada con-

troller em um arquivo em separado, inclusive podemos agrupá-los dentro de uma pasta chamada controllers. Vamos criar o arquivo public/js/controllers/ContatosControlller.js:

```
// public/js/controllers/ContatosController.js

angular.module('contatooh').controller('ContatosController',
  function() {

});
```

Nesse código, chamamos novamente a função angular.module, mas dessa vez passamos apenas o nome do módulo como parâmetro. A diferença é que, no lugar de criarmos um módulo, estamos acessando um já existente. Em seguida, encadeamos uma chamada à função **controller** para criarmos nosso primeiro controlador.

A função controller recebe como primeiro parâmetro o **nome** do controller, o segundo, a **função** que o de□nirá. Se o papel do controller é disponibilizar para a view o dado de que ela precisa, vamos criar a variável total:

```
// public/js/controllers/ContatosController.js

angular.module('contatooh').controller('ContatosController',
  function() {
    var total = 0;
});
```

Por □m, vamos importar o nosso controller imediatamente após a importação do nosso módulo principal em index.html:

```
<! -- public/index.html -->

...
<script src="vendor/angular/angular.js"></script>
<script src="js/main.js"></script>
<script src="js/controllers/ContatosController.js"></script>
...
```

O objeto especial $scope funciona da seguinte maneira: tudo o que for adicionando dinamicamente nele estará disponível na view. Mas como teremos acesso a esse objeto? É aí que entra o sistema de **injeção de dependência** do AngularJS.

Injeção de dependência

Podemos solicitar ao AngularJS o objeto $scope, adicionando-o como parâmetro da função que define nosso controller:

```
// public/js/controllers/ContatosController.js

angular.module('contatooh').controller('ContatosController',
  function($scope) {
    var total = 0;
});
```

O sistema de injeção de dependências do AngularJS é baseado no nome do parâmetro, logo, se usarmos qualquer outro nome, o framework não será capaz de injetá-lo em nosso controller.

gscope é um POJO

Temos o objeto que representa o escopo do nosso controller ContatosController e sabemos que ele é a cola entre o controller e a view. E agora? $scope é um *Plain Old Javascript Object* (POJO), logo, podemos adicionar propriedades dinamicamente e o mais fantástico é que elas serão acessíveis pela view através de AE.

Vamos substituir a declaração da variável total adicionando a propriedade de mesmo nome em nosso objeto $scope. Aproveitaremos para inicializar seu valor com zero:

```
// public/js/controllers/ContatosController.js

angular.module('contatooh').controller('ContatosController',
  function($scope) {
    $scope.total = 0;

});
```

Já podemos testar abrindo nossa página no navegador:

http://localhost:3000

Repare que agora o texto exibido é:

Contatos cadastrados: 0

Isso prova que nossa AE conseguiu buscar no escopo de ContatosController o dado de que precisava, isto é, no objeto $scope que representa o escopo do controller ContatosController.

Já temos view e controller, mas onde está o model do MVC? É o que veremos a seguir.

4.10 A□□□Z□JS u □□ou □o□ MVC

AngularJS não é rígido na escolha de um model, isto é, qualquer tipo literal do JavaScript ou objeto podem ser models, logo, disponíveis no escopo do controller.

Por exemplo, poderíamos ter:

```
// Object
$scope.contato = {
  nome: 'Flávio Henrique',
  sobrenome: 'Souza Almeida'
};

// boolean
$scope.exibir = true;

// Number
$scope.salario = 100.12;

// Date
$scope.admissao = new Date();
```

É por isso que agora chamaremos $scope.total de model. Falando em model, precisamos incrementá-lo toda vez que o usuário clicar no botão Novo. Para isso, utilizaremos mais uma diretiva do Angular.

4.11 Executando ações no controller através de diretiva

Como incrementar o total exibido quando o botão for clicado? É através da diretiva ng-click que conseguiremos este resultado:

```
<!-- public/index.html -->
...
<button ng-click="incrementa()">Novo</button>
...
```

A diretiva ng-click tem como valor a **chamada de uma função**, em nosso caso, incrementa(). Ela se comporta da mesma maneira como se tivéssemos associado a função ao evento onclick em JavaScript. Seguindo a mesma lógica, existem diretivas como ng-blur, ng-mouseover etc. O AngularJS possui uma para cada tipo de evento do JavaScript.

Mas onde nossa diretiva procurará a função incrementa()? No escopo de nosso controller, onde mais seria? Vamos adicionar em $scope a função que incrementará nosso total:

```
// public/js/controllers/ContatosController.js

angular.module('contatooh').controller('ContatosController',
  function($scope) {
    $scope.total = 0;
    $scope.incrementa = function() {
      $scope.total++;
    };

});
```

Nossa função, atribuída a $scope.incrementa simplesmente incrementa a propriedade $scope.total e será chamada toda vez que o botão de nossa página for clicado. Vamos visualizar a página e clicar algumas vezes no botão:

http://localhost:3000

C□□□□ou □Zl e□ □□f□u □ u □□□Z□ u □ □u □□i □-□□ou X h□□u □□ - □□ou

Você já deve ter percebido a diferença entre <%= >e {{ }}. O primeiro, é uma expressão avaliada no lado do servidor pelo Express. Depois de processada, o que é enviado para o navegador é o HTML já mesclado, isto é, com a informação já substituída.

A AE, por outro lado, é avaliada no navegador. Você pode até exibir o código-fonte da nossa página. Nele, você verá que a AE {{total}} continua lá, mas, se você inspecionar o elemento do DOM que carrega a AE, o que é exibido é '0', indicando que o AngularJS processou a AE modi□cando o DOM no lado do cliente.

Aprenderemos na seção 5.4 como carregar os dados do servidor para alimentar nossos templates no lado do cliente.

Agora que já sabemos um pouco mais de AngularJS, já podemos iniciar nossos estudos sobre seu sistema de rotas na próxima seção.

4.13 S□□□□u □Z□u Z□□□□hZ□ □□□ (SPA) u □□□□□ □□□-□u □ □

Nossa aplicação Contatooh por enquanto possui apenas a página public/index.html. Além dela, mais duas serão necessárias: uma que lista contatos e outra que exibe um contato selecionado da lista. Faremos isso usando a estratégia **Single Page Application**.

Single Page Application (SPA)

Single Page Application (SPA) é uma aplicação entregue para o navegador que não recarrega a página durante seu uso. Aplicações deste tipo tendem a dar uma experiência mais □uida para os usuários, ao mesmo tempo em que favorece o servidor, enviando uma quantidade de dados menor para ser processada.

Em SPA, a página principal, por exemplo, index.html, é carregada

de atualizar a página principal com o conteúdo das parciais:

Exemplos de URLs

http://localhost:3000/index.html#/contatos
http://localhost:3000/index.html#/produtos
http://localhost:3000/index.html#/contato/1

O AngularJS possui seu próprio sistema de rotas. É o que veremos a seguir.

4.14 O ⊠o⊠ ⊠R⊠ u

AngularJS possui um sistema de rotas que visa blindar o desenvolvedor da complexidade pela atualização de áreas da página utilizando Ajax, mais uma vez, evitando que ele manipule o DOM diretamente.

Necessidade de um servidor

Um ponto importantíssimo é que **o sistema de rotas do AngularJS só funcionará se você estiver rodando sua aplicação num servidor web** (o que não é um problema para nós), mesmo que você esteja apenas querendo testá-lo. Isso é necessário porque o AngularJS dispara requisições Ajax para buscar do servidor parciais que ele precisa exibir.

Antes da versão 1.2, o AngularJS já vinha com o sistema de rotas, mas a equipe decidiu movê-lo do core para um módulo em separado chamado **ngRoute**. Precisamos baixá-lo através do Bower na linha de comando:

```
bower install angular-route#1.3 --save
```

Não podemos nos esquecer de importar o script em nossa página public/index.html imediatamente após a importação do script do AngularJS:

```
<!-- public/index.html -->
...
<script src="vendor/angular/angular.js"></script>
<script src="vendor/angular-route/angular-route.js"></script>
```

```
<script src="js/main.js"></script>
<script src="js/controllers/ContatosController.js"></script>
```

Isso ainda não é suficiente, precisamos tornar nosso módulo principal, contatooh, ciente deste módulo. Para isso, editaremos public/js/main.js adicionando o módulo **ngRoute** como sua dependência. Isso é importante, caso contrário, não teremos acesso à artefatos injetáveis deste módulo. Vamos fazer isso adicionando-o no array passando como segundo parâmetro da função angular.module:

```
// public/js/main.js

angular.module('contatooh', ['ngRoute']);
```

Sistema de rotas ativado, porém não configurado! Antes de configurá-lo, vamos alterar a página public/index.html substituindo o conteúdo da tag <body> por uma com a diretiva ng-view, mas mantendo a importação dos scripts.

Não podemos esquecer de **remover** a diretiva ng-controller de <body>, o que é necessário porque a associação do controller será feita através da configuração de nossas rotas. Isso permite que parciais possam utilizar controllers diferentes de acordo com quem as usa.

Nossa página deve ficar assim:

```
<!-- public/index.html -->
...
<body>
  <div ng-view></div>
  <script src="vendor/angular/angular.js"></script>
  <script src="vendor/angular-route/angular-route.js"></script>
  <script src="js/main.js"></script>
  <script src="js/controllers/ContatosController.js"></script>
</body>
...
```

A diretiva ng-view sinaliza para o sistema de rotas a área da página que receberá **views parciais**. A diferença de uma view parcial para uma view como a index.html é que a primeira não possui as tags <html>, <head>

e <body>, logo, para serem exibidas, precisam ser incluídas dinamicamente dentro de uma página com a diretiva ng-view, em nosso caso, a página index.html.

4.15 C□□Z□o□□u□□Z□h□Z□□

Temos nossa view principal index.html pronta, agora precisamos criar nossas primeiras parciais que popularão a diretiva ng-view. Vamos criar a pasta app/public/partials e dentro dela guardar todas as nossas views parciais.

> **U□Z o□□u□□□, o□□u□u □□ u□Z□□u□u □□ Zl□u□**
>
> No lugar de criarmos uma com a diretiva ng-view, poderíamos utilizar a tag <ng-view></ng-view>. Lembre-se que diretivas do AngularJS podem ser utilizadas como tags, atributos e até mesmo como comentário! Para saber quais formatos estão disponíveis, precisamos consultar a documentação do AngularJS.

Primeiro criaremos a view parcial que será responsável pela listagem dos contatos. Repare que movemos para ela nosso botão e o parágrafo com a Angular Expression:

```
<!-- public/partials/contatos.html -->
<h1>Bem-vindo ao Contatooh</h1>
<button ng-click="incrementa()">Novo</button>
<p>Contatos cadastrados: {{total}}</p>
```

A nossa view, ainda incompleta, exibirá o contato da lista.

```
<!-- public/partials/contato.html -->

<h1>Contato</h1>
```

Agora temos três arquivos:

- **public/index.html**: página principal, aquela que receberá views parciais de acordo com as rotas da aplicação.
- **public/partials/contatos.html**: view parcial responsável pela listagem de contatos.
- **public/partials/contato.html**: view parcial responsável pela exibição do contato selecionado da lista.

Se abrirmos a página no navegador apenas veremos uma página em branco. Precisamos configurar as rotas da aplicação e definir que a URL http://localhost:3000/index.html#/contatos deve carregar a view parcial public/partials/contatos.html. Preparado?

4.16 Configurando as rotas

A configuração das rotas da aplicação costuma ser feita no módulo principal da aplicação, ou seja, no arquivo main.js. Utilizamos a função config, que recebe uma função que tem como parâmetro um artefato injetado pelo AngularJS responsável pela criação de rotas, o objeto $routeProvider. Caso não tivéssemos importado o módulo ngRoute, ele não estaria disponível para injeção:

```
// public/js/main.js

angular.module('contatoos', ['ngRoute'])
  .config(function($routeProvider) {

    $routeProvider.when('/contatos', {
      templateUrl : 'partials/contatos.html',
      controller:   'ContatosController'
    });
});
```

O objeto $routeProvider possui a função when. Nela informamos a rota (sem o #) e no segundo parâmetro um objeto que define qual **template** (parcial) será carregado para a rota e qual será seu **controller** através das propriedades templateURL e controller, respectivamente.

Um ponto importante é que toda rota configurada por $routeProvider para ser disparada deve ser acessada através da URL da página principal adicionando o prefixo #, o famoso *hash*. Por exemplo:

http://localhost:3000/index.html#/contatos

Como nossa página index.html está dentro da pasta public podemos acessá-la diretamente através da URL /, é por isso que também podemos disparar a rota do AngularJS desta maneira mais enxuta. Podemos testar:

http://localhost:3000/#/contatos

Fig. 4.6: Testando rota k/contatos

Não podemos esquecer do #, porque se acessássemos a URL (http://localhost:3000/contatos) a lista de contatos do servidor seria devolvida para nós. Não queremos disparar uma rota do servidor, queremos disparar uma rota no lado do cliente que saberá atualizar nossa página index.html já carregada no navegador com determinada parcial.

Mais uma rota

Precisamos criar a rota responsável pela exibição da parcial partials/contato.html. Esta página exibirá o contato selecionado da lista, logo, precisa receber como parâmetro o id do contato selecionado:

```
$routeProvider.when('/contato/:contatoId', {
  templateUrl : 'partials/contato.html',
  controller: 'ContatoController'
});
```

Repare que a rota é um pouco diferente terminando com o curinga :contatoId. É através deste nome que dentro do controller ContatoController podemos referenciar o id do contato passado na URL, por exemplo:

htttp://localhost:3000/index.html#/contato/1

Mas como obtê-lo? É o que veremos adiante.

4.17 O f u g u PZZ

Vimos na seção anterior a necessidade de obtermos o ID do contato a partir de uma rota do AngularJS. Para isso, criaremos o controller public/js/controllers/ContatoController.js:

```
// public/js/controllers/ContatoController.js

angular.module('contatooh').controller('ContatoController',
  function($scope, $routeParams) {

    console.log($routeParams.contatoId);

});
```

Repare que ContatoController recebe via injeção o objeto $routeParams. Este objeto só está disponível porque estamos utilizado o módulo ngRoute.

É acessando a propriedade $routeParams.contatoId que temos acesso ao parâmetro passado via URL. Por enquanto, apenas o imprimiremos no console, mas no futuro , precisaremos buscar o contato associado a este ID em nosso servidor Express, que por sua vez o buscará no banco de dados para, no m, ser exibido em nossa parcial.

Não podemos nos esquecer de importar ContatoController em public/index.html:

```
<!-- public/index.html -->
...
```

```
<body>
  <h1>Bem-vindo ao Contatooh</h1>
  <div ng-view></div>
  <script src="vendor/angular/angular.js"></script>
  <script src="vendor/angular-route/angular-route.js"></script>
  <script src="js/main.js"></script>
  <script src="js/controllers/ContatosController.js"></script>
  <script src="js/controllers/ContatoController.js"></script>
</body>
...
```

Podemos testar:
http://localhost:3000/k/contato/1

Fig. 4.7: Testando rota k/contato/1

Mas se a rota acessada não existir? Podemos adicionar uma rota padrão.

Adicionando rota padrão

Por fim, podemos adicionar uma rota padrão caso o endereço da rota não exista. Fazemos isso através da função $routeProvider.otherwise. Nela, passamos um objeto com a propriedade redirectTo, que aponta para um rota alternativa, em nosso caso, aquela que lista os contatos:

```
$routeProvider.otherwise({redirectTo: '/contatos'});
```

Agora que já entendemos como funciona o mecanismo de rotas do AngularJS precisamos concluir nossas parciais. Vamos fazer isso com uma "ajudinha" do Bootstrap para depois voltarmos para o AngularJS.

- Estilo visual base pra maioria das tags
- Ícones através do *glyphicon*
- Grids prontos para uso
- Componentes CSS
- Plugins JavaScript
- Tudo responsivo e *mobile-first*

Baixando o Bootstrap através do Bower

Baixamos o AngularJS através do Bower e agora faremos a mesma coisa com o Bootstrap. Para isso, vamos executar o comando a seguir dentro da pasta raiz do projeto:

```
$ bower install bootstrap --save

bower not-cached git://github.com/twbs/bootstrap.git#3.3
bower resolve git://github.com/twbs/bootstrap.git#3.3
bower download https://github.com/twbs/bootstrap/archive/
                                                    v3.3.1.tar.gz
...
bower resolved git://github.com/twbs/bootstrap.git#3.3.1
bower not-cached git://github.com/jquery/jquery.git#>= 1.9.1
bower resolve git://github.com/jquery/jquery.git#>= 1.9.1
bower download https://github.com/jquery/jquery/archive/
                                                    2.1.1.tar.gz
bower extract jquery#>= 1.9.1 archive.tar.gz
bower resolved git://github.com/jquery/jquery.git#2.1.1
bower install bootstrap#3.3.1
bower install jquery#2.1.1

bootstrap#3.3.1 public/vendor/bootstrap
 jquery#2.1.1

jquery#2.1.1 public/vendor/jquery
```

Um ponto curioso é que o Bower não baixou apenas o Bootstrap para nós, mas também o jQuery (http://jquery.com) . Isso acontece porque o Bootstrap tem como dependência o jQuery, utilizado por alguns de seus componentes. Agora falta integrar o Boostrap com nossas views.

Adicionando o Bootstrap em nossas páginas

O CSS do Bootstrap precisa ser importado. Como estamos criando uma SPA, sua importação é na página principal, em nosso caso, public/index.html.

```
<!-- public/index.html -->
...
<head>
  <meta charset="UTF-8">
  <title>Contatooh</title>
  <link rel="stylesheet"
    href="vendor/bootstrap/dist/css/bootstrap.css">
  <link rel="stylesheet"
    href="vendor/bootstrap/dist/css/bootstrap-theme.css">
</head>
```

Aproveite a adicione a classe container do Bootstrap na tag body:

```
...
<body class="container">
...
```

A classe container fará com que todo o conteúdo da página que alinhado ao centro. Você já pode visualizar o resultado, inclusive pode adicionar e remover a classe para ver a diferença (visualize em tela cheia para facilitar a percepção):

http://localhost:3000/k/contatos

Como todas as parciais serão adicionadas dentro da com a diretiva ng-view lha de body, todas serão centralizadas.

Agora vamos completar a página public/partials/contatos.html:

```
<!-- public/partials/contatos.html -->
```

```
<div class="jumbotron">
  <h1 class="text-center">
    Bem-vindo ao Contatooh
  </h1>
</div>
<button class="btn btn-primary" ng-click="incrementa()">
  Novo
</button>
<p>Contatos cadastrados: {{total}}</p>
<div class="table-responsive">
  <table class="table table-hover">
    <tr>
      <th>NOME</th>
      <th>E-MAIL</th>
      <th class="text-center">Ação</th>
    </tr>
    <tr>
      <td>
        <a>Contato</a>
      </td>
      <td>cont@empresa.com.br</td>
      <td class="text-center">
        <button class="btn btn-warning">
          Remover
        </buttom>
      </td>
    </tr>
  </table>
</div>
```

Repare que usamos uma série de classes do Bootstrap, são elas:

- btn: classe que fornece um visual diferenciado para o botão.
- btn-primary: destaca o botão (azulado e maior) quando combinada com a classe btn.
- btn-warning: destaca o botão, sem aumentar seu tamanho colocando-o na cor laranja.

Com AngularJS, a coisa muda um pouco: o AngularJS pede ao servidor os dados de que precisa e se encarrega de atualizar a view com esses dados no lado do cliente, o famoso data binding. Já sabemos que qualquer dado "pendurado" no objeto $scope é visível pela view, sendo assim, vamos criar uma lista de contatos no escopo de ContatosController:

```
// public/js/controllers/ContatosController.js

angular.module('contatooh').controller('ContatosController',
  function($scope) {

    // código omitido

    $scope.contatos = [
      {
        "_id": 1,
        "nome": "Contato Angular 1",
        "email": "cont1@empresa.com.br"
      },
      {
        "_id": 2,
        "nome": "Contato Angular 2",
        "email": "cont2@empresa.com.br"
      },
      {
        "_id": 3,
        "nome": "Contato Angular 3",
        "email": "cont3@empresa.com.br"
      }
    ];
});
```

Muito bem, temos uma lista e sabemos que nossa parcial contatos.html precisa ser construída a partir dela, mas como? Apelaremos para mais uma diretiva do AngularJS, a ng-repeat.

Essa diretiva permite que o AngularJS repita a criação de um elemento de nossa página a partir dos dados de um array. Em nosso caso, vamos adicioná-la na <tr> que exibe os dados do contato:

```
<!-- partials/contatos.html -->
...
  <tr ng-repeat="contato in contatos">
    <td>
      <a>{{contato.nome}}</a>
    </td>
    <td>{{contato.email}}</td>
    <td class="text-center">
      <button class="btn btn-warning">
        Remover
      </buttom>
    </td>
  </tr>
...
```

A diretiva possui um valor especial:

ng-repeat="contato in contatos"

Nela, indicamos qual lista estamos varrendo, em nosso caso contatos. Mas como ter acesso a cada contato da lista? Para isso, damos um apelido para cada item da lista, em nosso caso, escolhemos o apelido contato.

Como temos três contatos em nossa lista, a diretiva ng-repeat será repetida três vezes e através da AE podemos ter acesso aos dados do contato, por exemplo, com {{contato.nome}} acessamos seu nome. Visualizemos o resultado:

http://localhost:3000/#/contatos

Fig. 4.10: A diretiva ng-repeat em ação

A diretiva ng-repeat é bem esperta, permitindo ainda filtrar nossa lista com pouquíssimo esforço. É o que veremos a seguir.

4.20 A diretiva ng-model e filtragem da lista

Podemos filtrar nossa lista de contatos, mas, primeiro, precisamos adicionar um input em nossa view parcial contatos.html:

```
<button class="btn btn-primary" ng-click="incrementa()">
  Novo
</button>
<input type="search" placeholder="parte do nome">
<p>Contatos cadastrados: {{total}}</p>
```

Não podemos simplesmente utilizar uma AE {{}} em nosso input, porque toda AE é somente leitura. Em nosso caso, queremos ler e gravar em uma propriedade no escopo do controller, isto é, queremos fazer **two-way data binding**. Para isso, usamos a diretiva ng-model:

```
<input ng-model="filtro" type="search"
  placeholder="parte do nome">
```

Agora, criamos a propriedade no escopo de ContatosController:

Ativação do AngularJS em nossa página

Vimos que o primeiro passo para tornarmos uma página gerenciada pelo AngularJS é importar seu script e adicionar a diretiva ng-app na tag que delimitará a área de atuação do framework, normalmente a tag html. Diretivas se parecem com atributos ou tags, com a diferença de que são processadas pelo AngularJS, adicionando novos comportamentos à página:

```
<html ng-app="contatooh">
```

Criação do módulo principal da aplicação

A diretiva ng-app aponta para o nome módulo principal da aplicação criado através do objeto angular, disponível globalmente pelo framework, chamando a função angular.module. Ela recebe o nome do módulo, em nosso caso, contatooh, e um array com suas dependências, que por enquanto deixamos vazio. A criação do módulo e outras configurações são feitas no arquivo main.js, mas nada impede que outro seja utilizado:

```
angular.module('contatooh', []);
```

Angular Expression

Aprendemos que página é view e que ela pode ter uma lacuna definida através de uma Angular Expression (AE) caracterizada por {{ }} e tudo que estiver dentro dela precisa ser buscado de algum lugar, desta forma, views atuam como templates que precisam de dados fornecidos por controllers:

```
<p>Contatos cadastrados: {{total}}</p>
```

Associação da view com um controller

Um controller é associado ao elemento que utiliza AE ou qualquer elemento pai do qual ele faça parte através da diretiva ng-controller, que possui como valor o nome do controller criado:

```
<body ng-controller="ContatosController">
```

Definição de um controller

Não é incomum criar controllers em arquivos separados iniciando com letra maiúscula e definidos através da função controller que recebe como parâmetro o nome do controller e a função que o define. Esta última, nos dá acesso ao objeto $scope, aquele que é a "cola" entre o controller e a view. Qualquer propriedade adicionada em scope é acessível através de AE no controller:

```
angular.module('contatooh').controller('ContatosController',
  function($scope) {

    $scope.total = 0;
});
```

Injeção de dependências

Também foi apresentado que o acesso ao objeto $scope é feito através do sistema de injeção de dependências do AngularJS, dependente do nome do parâmetro. Não precisamos criar o objeto, apenas indicar que precisamos dele.

A diretiva ng-click

Por fim, vimos que há diretivas que permitem executar ações definidas no escopo do controller como a diretiva ng-click:

```
<button ng-click="incrementa()">Novo</button>
```

Nesta diretiva, evocamos uma função que, quando presente no controller, será executada. Durante sua execução podemos alterar dados "pendurados" em $scope que o AngularJS se encarregará de sincronizá-lo com a view:

```
angular.module('contatooh').controller('ContatosController',
  function($scope) {

    $scope.total = 0;

    $scope.incrementa = function() {
```

```
        $scope.total++;
    };
});
```

Esse processo é chamado de one-way data binding, que permite apenas ler dados no escopo do controller.

A diretiva ng-repeat

Outro diretiva poderosa que vimos foi a ng-repeat, que permite repetir o elemento no qual ela foi atribuída com base em um array de nido no escopo do controller:

```
<tr ng-repeat="contato in contatos">
    <td>
      <a>{{contato.nome}}</a>
    </td>
    <td>{{contato.email}}</td>
    ...
  </tr>
...
```

ng-repeat e filtro

Inclusive, vimos que podemos ltrar a lista adicionando | filter ao nal do valor da diretiva:

```
<tr ng-repeat="contato in contatos | filter: filtro">
```

ng-model e two-way data binding

O dado filtro, diferente de AE e da diretiva ng-repeat, que apenas lêem dados do escopo de um controller, permite também alterar, o que caracteriza o two-way data binding. Esta forma de associaç ão é feita através da diretiva ng-model:

```
<input ng-model="filtro" type="search"
 placeholder="parte do nome">
```

Agora que já entendemos os mecanismos básicos do AngularJS, no próximo capítulo faremos sua integração com nosso servidor Express.

CZ£ –¶ 5

Integrando AngularJS e Express

"O homem tem mil planos para si mesmo. O destino, só um para cada."
– Mêncio

No capítulo anterior, vimos desde a conguração do AngularJS em nosso projeto até o uso de diretivas para ampliar as capacidades do navegador, inclusive como o AngularJS mantém nosso model sincronizado com nossa view através de *data binding*. Porém, não realizamos nenhuma integração entre front-end e back-end. É justamente isso que faremos neste capítulo.

5.1 O u g

Nosso front-end atualmente trabalha com dados estáticos denidos em nossos controllers. A ideia agora é buscar esses dados diretamente de nosso servidor.

Não precisamos nos preocupar com o back-end, pois já criamos serviços que retornam dados no formato JSON. O que falta é a "cola" entre nosso front-end e nosso back-end.

O AngularJS possui o serviço **ghttp** responsável por requisições Ajax. Injetamos este serviço em nossos controllers como qualquer outro artefato do framework:

```
angular.module('contatooh').controller('ContatosController',
    function($scope, $http) {

});
```

I l e ou ou u ox hZ f Z u ZoZ u u

Não importa a ordem dos parâmetros do nosso controller, são os nomes dos parâmetros que importam. Lembre-se que o sistema de injeção de dependências do AngularJS é baseado em nomes, logo, a declaração a seguir funciona perfeitamente, mesmo recebendo primeiro $http:

```
angular.module('contatooh').controller('ContatosController',
    function($http, $scope) {});
```

O serviço $http recebe como parâmetro um objeto com as configurações da requisição.

```
{
    method: /* método utilizado, pode ser GET, POST, PUT ou
               DELETE */
    url: /* endereço do recurso acessado */
    data: /* objeto no qual cada propriedade será um parâmetro
    na requisição */
}
```

Para obtermos a lista de contatos de nosso servidor através de uma requisição do tipo GET, configuramos o serviço da seguinte maneira:

```
$http({method: 'GET', url: '/contatos'});
```

Repare que a propriedade data foi omitida porque não queremos enviar parâmetros na requisição.

> AngularJS possui atalhos para os métodos GETe POSTatravés das funções $http.get('url') e $http.post('url').

Mas como ter acesso ao resultado da requisição? Para podemos entender com clareza como o $http funciona, precisamos revisitar uma das formas mais utilizadas: o uso de *callbacks*.

5.2 P□□□Z□Zl e□ Z□□□h□□Z u hZ□□fZh□ HELL

Toda requisição Ajax é assíncrona por natureza e nunca sabemos quando ela será concluída de fato. É por isso que uma estratégia comum é passar para essas funções um **callback**, aquela função que guarda nossa lógica que será chamada pela função assíncrona assim que ela for concluída. Inclusive, podemos capturar no callback o resultado da função.

Vejamos um exemplo no qual realizaremos cinco ações assíncronas na sequência. A primeira busca os contatos através de uma requisição com $http. Logo em seguida, eles são exibidos e modi□cados. Depois, os dados são enviados novamente para o servidor. Por □m, exibimos uma mensagem para o usuário indicando que a operação foi bem-sucedida:

```
function exibeContatos(contatos, callback) {
    /*
        Exibe os contatos na tela e logo em seguida
        passa os contatos para a função de callback
    */

    callback(contatos);
}

function modificaContatos(contatos, callback) {
    /*
```

```
            Modifica os contatos seguindo algum critério
            e depois chama o callback com contatos modificados.
        */

        callback(contatos);
}

function atualizaContatos(contatos, callback) {
        /*
            Recebe os contatos modificados e
            envia novamente para o servidor para que
            sejam gravados. No final, chama o callback
        */

        callback(contatos);
}

/* A seguir, a "pyramid of DOOM" ou ninho de amafagafa */

$http.get('/contatos', function(contatos) {
    exibeContatos(contatos, function(contatos) {
            modificaContatos(contatos, function(contatosModificados) {
                atualizaContatos(contatosModificados,
                function(contatos) {
                    $scope.mensagem ={
                        texto:  'Contatos atualizados com sucesso'
                    };
                });
            });
    });
});
```

Você não precisa meditar durante muito tempo sobre o código anterior para ver que sua legibilidade não é uma das melhores e que todas as nossas funções foram obrigadas a receber um callback, o que polui sua interface de uso. Existe até um nome para isso: **callback HELL**.

Callback HELL

O callback HELL é um problema que aparece na programação assíncrona quando precisamos executar códigos assíncronos ordenadamente. Para conseguirmos o resultado desejado, precisamos aninhar callbacks, o que compromete legibilidade e dificulta manutenção do nosso código.

Pyramid of DOOM

O callback HELL visualmente é caracterizado pela **pyramid of DOOM**, um triângulo deitado que fica evidente com o aninhamento das funções de callback.

Existe ainda outro problema que tange o tratamento de erros. Usar um `try` e `catch` não funcionará em nosso código:

```
/* não funciona */
try {
  $http.get('/contatos', function(contatos) {
    exibeContatos(contatos, function(contatos) {
      modificaContatos(contatos, function (contatosModificados) {
        atualizaContatos(contatosModificados,
        function(contatos) {
          $scope.mensagem= {
            texto: 'Contatos atualizados com sucesso'
          };
        });
      });
    });
  });
} catch(erro) {  /* nunca é chamado */
    /* trata o erro */
};
```

Como o código é assíncrono, ele não estará na pilha no momento em que o `try` for avaliado.

São por esses motivos que `$http` não trabalha da forma como vimos. Ele utiliza um padrão bem definido que veremos a seguir.

5.3 P□□□□□u □: h□□f Z□ u □o□ □ hZ□□f Zh□ HELL

O $http, no lugar de receber um callback diretamente em sua chamada, trabalha um pouco diferente: ela devolve um valor.

```
var contatos = $http({method: 'GET', url: '/contatos'});
```

Espere um pouco! Se $http devolve um valor que normalmente seria acessível apenas em uma função de callback passada como parâmetro, isso signi□ca que sua chamada é síncrona? Sua execução bloqueia a leitura do nosso código até que seja concluída?

Não exatamente. O exemplo a seguir exibe no console o texto antes de concluir a requisição:

```
var contatos = $http({method: 'GET', url: '/contatos'});
console.log("total de contatos " + contatos.length);
// exibe undefined
```

Se o código anterior tem problema, por que então o $http, em vez de retornar a lista de contatos do servidor, não faz isso dentro de uma função de callback passada como parâmetro? Mesmo que caiamos no callback HELL, sabemos que nosso código funcionará.

O padrão Promise

O primeiro passo para entender este mistério é saber que $http não retorna a lista de contatos, mas uma **promise** (promessa) de que ele tentará buscar esses dados:

```
var promise = $http({method: 'GET', url: '/contatos' });
```

Uma promise é um objeto que fornecerá o resultado futuro de uma ação. No exemplo anterior, como estamos executando uma requisição assíncrona, não sabemos quando ela nos devolverá seu resultado, sendo assim, □camos com a **promessa** de sua devolução, sua promise.

Fazendo uma analogia: quando alguém nos promete algo, temos apenas sua promessa, pois não sabemos quando ela será cumprida. Independente de ela serr cumprida ou não, tocamos nossa vida. Porém, sabemos o que pode

acontecer se a promessa for cumprida ou não. É por isso que uma *promise* possui estados.

Estados de uma promise

Uma *promise* possui três estados e dependendo desses estados, ações são executadas:

- **fulfilled**: quando a *promise* é bem-sucedida;
- **rejected**: quando a *promise* é rejeitada;
- **failed**: quando não é nem bem-sucedida nem rejeitada.

Uma *promise* que foi fullfilled ou rejected não pode ser fullfilled ou rejected novamente.

As funções then e catch

Uma *promise* possui o método **then**, que recebe como parâmetros call-backs. O primeiro é executado quando o status da promise for fulfilled; o segundo, para os estados rejected e failed.

```
function obterDados(retorno) {
    /* faz algo com o retorno */
}

var promise = $http.get('/contatos');
promise
    .then(obterDados, function(erro) {
        console.log(erro.status)
        console.log(erro.statusText)
    });
);
```

Porém, o AngularJS introduz a função catch, que permite isolar o call-back dos estados rejected e failed:

```
function obterDados(retorno) {
    /* faz algo com o retorno */
```

```
}

var promise = $http.get('/contatos');
promise
    .then(obterDados)
    .catch(erro) {
        console.log(erro.status)
        console.log(erro.statusText)
    }
);
```

Nesse exemplo, através do parâmetro retorno, temos acesso a propriedades especiais que nos permitem acessar os dados retornados, inclusive obter mensagens de erro enviadas pelo servidor:

- **data**: o *body* da resposta transformado e pronto para usar;
- **status**: número que indica o status HTTP da resposta;
- **statusText**: texto HTTP da resposta.

Ainda é possível ter acesso ao objeto header e config, este último com as configurações utilizadas na requisição.

Encadeamento da função then

Podemos encadear a chamada à função then, garantindo que nosso código execute na sequência. Vejamos nosso exemplo, antes dominado pelo *callback HELL* e agora utilizando *promises*:

```
function exibeContatos(contatos) {
    /*
        Exibe os contatos na tela
    */

    return contatos;
}

function modificaContatos(contatos) {
```

```
    /*
        Modifica os contatos seguindo algum critério
    */

    return contatos;
}

function atualizaContatos(contatos) {
    /*
        Recebe os contatos modificados e
        envia novamente para o servidor para que
        sejam gravados.
    */

    return contatos;
}

var promise = $http.get('/contatos');
promise
    .then(exibeContatos)
    .then(modificaContatos)
    .then(AtualizaContatos)
    .then(function(contatos) {
        $scope.mensagem=
            {texto: 'Contatos atualizados com sucesso'};
    })
    .catch(erro) {      /* se algo der errado, trata */
        console.log(erro.status)
        console.log(erro.statusText)
    }
);
```

Repare que cada função não precisou receber um callback como parâmetro; utilizamos até um retorno normal de função. Este retorno estará disponível para a próxima chamada à função then.

Tratamento de erro

A *promise* permite tratar erros em um único local, dentro da função **catch**. Se uma *promise* for rejected, failed ou alguma exceção for lançada, temos um único local para tratar esses erros.

As funções success e error

O objeto $http adiciona convenientemente em suas *promises* as funções **success** e **error**. A primeira recebe o callback que será executado quando a *promise* for fulfilled. Já a segunda, o callback que será executado quando a *promise* for rejected ou failed.

```
var promise = $http.get('/contatos');
promise
.success(function(data, status, headers, config) {
    $scope.contatos = data;
})
.error(function(statusText, status, headers, config) {
    console.log(statusText);
    console.log(status);
});
```

Ainda podemos omitir a declaração da variável e omitir os parâmetros que não nos interessam:

```
$http.get('/contatos')
.success(function(data) {
    $scope.contatos = data;
})
.error(function(statusText) {
    console.log(statusText);
});
```

g□□ □ □□. g.Z□Z□

Para aqueles que utilizam jQuery, o $http possui semelhanças com $.ajax. Ambos fornecem uma API de baixo nível para realizar requisições Ajax. Apesar de cada objeto ter suas funções específicas para a tarefa, a grande diferença é a integração de $http com o ciclo de vida do AngularJS, também chamado de **digest cycle**.

$http notificará o framework sempre que a requisição assíncrona alterar algum model, garantindo a sincronização entre model e view. Se por acaso você utilizar o $.ajax ou qualquer outra função especializada do jQuey com AngularJS, será você o responsável em avisar o framework das modificações realizadas. chamando $scope.$apply():

```
$.ajax({
    url: 'enderecoDoServico',
    success: function(data, status, xhr) {
        // atualiza informação em $scope
        $scope.dadoQualquer = data;

        // notifica o AngularJS da modificação
        $scope.$apply();
    }
})
```

Quando chamar $scope.$apply(), lembre-se que você estará assumindo a responsabilidade de sincronizar a view com o model, logo, é altamente recomendado que você utilize $http ou outro artefato do AngularJS quando necessário.

Com o padrão *promise*, nosso código fica mais legível, pois ganha cara de código síncrono, mesmo que tenhamos uma ou mais funções assíncronas. Não precisamos nem nos preocupar se a função é síncrona ou assíncrona, pois teremos a garantia de que todas serão executadas na ordem em que as chamarmos através da função then.

O tema *promise* é bem amplo e por agora o que sabemos é suficiente para continuarmos nossa caminhada com AngularJS. Caso você queira saber mais

sobre o assunto, consulte a especi□cação o□cial em http://promises-aplus.github.io/promises-spec/.

5.4 Of□u□o□h□□□Z□□□h□□ g□□□□

Agora que você aprendeu que $http retorna uma *promise* e como executar um código quando ela for bem-sucedida, já podemos refatorar nosso **ContatosController** para deixar de trabalhar com dados estáticos e obtê-los do nosso back-end feito com Express:

```
// public/js/controllers/ContatosController.js

angular.module('contatooh').controller('ContatosController',
    function($scope, $http) {

        $scope.contatos = [];

        $scope.total = 0;

        $scope.filtro = '';

        $scope.incrementa = function() {
            $scope.total++;
        };

        $http.get('/contatos')
        .success(function(data) {
            $scope.contatos = data;
        })
        .error(function(statusText) {
            console.log("Não foi possível obter a lista de
                        contatos");
            console.log(statusText);
        });
});
```

Certi□que-se de que o servidor esteja rodando e veja o resultado. Tudo deve continuar funcionando.

A única diferença é que nossos dados vieram de nosso servidor. Aliás, você pode consultar o console caso alguma coisa tenha saído errada. Não se preocupe ainda com uma mensagem amigável para o usuário, atacaremos esse problema mais tarde.

Refatorando a exibição do total de contatos cadastrados

Podemos aproveitar e atualizar nossa lógica que exibe o total de contatos cadastrados. Vamos remover a função $scope.incrementa e o atributo $scope.total. Agora, via AE, exibimos o tamanho do array $scope.contatos diretamente em nossa view:

```
<!-- public/partials/contatos.html -->
<p>Contatos cadastrados: {{contatos.length}}</p>
```

Toda vez que novos contatos forem adicionados ou removidos, nossa view exibirá dedignamente o total de contatos por causa do *data binding*.

ghttp? Existe algo mais abstrato?

Poderíamos usar $http para realizar a "cola" entre nosso front-end e nosso back-end, porém, como utilizamos o padrão REST em nosso servidor, podemos utilizar um objeto especializado e de mais alto nível para consumir nossos *REST Endpoints*.

5.5 O o R u hu : Z f Z Z o u Z n

Vimos como realizar requisições Ajax, assíncronas por natureza, através do serviço $http e como ele utiliza o padrão promise para ajudar na legibilidade e manutenção de nosso código, evitando assim o *callback HELL*.

Apesar de $http ser funcional, AngularJS possui um serviço de mais alto nível chamado **gresource**, específico para consumir REST Endpoints! Isso é excelente, uma vez que nossas rotas criadas no Express seguem este padrão.

O serviço $resource não faz parte do core do AngularJS, logo, precisamos do módulo ngResource para que ele esteja disponível para nós. Isso não é um problema, podemos baixá-lo através do bower:

```
bower install angular-resource#1.3 --save
```

Não podemos esquecer de importá-lo em nossa página index.html, logo após a importação do módulo ngRoute:

```
<script src="vendor/angular-resource/angular-resource.js">
</script>
```

Você deve se lembrar que isso ainda não é suficiente e que ainda precisamos indicar que nossa aplicação fará uso do módulo em nosso arquivo main.js:

```
// public/js/main.js
var app = angular.module('contatooh',['ngRoute', 'ngResource']);
```

Repare que agora nossa aplicação depende de dois módulos: ngRoute e ngResource, ambos definidos no array passado como parâmetro para a função angular.module.

Até aqui, toda burocracia a respeito da importação do módulo ngResource já foi atendida. Em ContatosController, precisamos substituir $http por $resource, artefato disponibilizado pelo módulo ngRoute:

```
angular.module('contatooh').controller('ContatosController',
    function($scope, $resource) {
    // código existente omitido
}
```

Muito bem! Substituímos $http por $resource para podermos trabalhar em alto nível com nossos REST Endpoints. Porém, ainda falta alterar a lógica de nosso controller para que ela faça uso de $resource.

5.6 Consumindo REST Endpoints com gresource

Agora que temos injetado $resource em nosso controller, podemos obter uma referência para o recurso contatos em nosso servidor. Para isso, passamos para $resource o identificador único deste recurso, em nosso caso, /contatos:

```
var Contato = $resource('/contatos');
```

O serviço $resource nos devolve um objeto que permite realizar uma série de operações seguindo o padrão REST para o recurso /contatos. O nome da variável está em maiúscula, algo intencional para diferenciá-lo de uma possível variável que represente o model para contato.

Nosso objetivo é realizar uma consulta ao nosso serviço e obter a lista de contatos. Para esta finalidade, existe a função query. A função, por debaixo dos panos, monta uma requisição do tipo GET para o recurso /contatos em nosso servidor. E como estamos trabalhando com uma requisição assíncrona, podemos pedir que ela nos devolva uma *promise*:

```
var promise = Contato.query().$promise;
```

O trâmite a seguir já é conhecido. Sabemos que uma *promise* possui as funções then e catch.

```
promise
  .then(function(contatos) {
    $scope.contatos = contatos;
  })
  .catch(function(erro) {
      console.log("Não foi possível obter a lista de contatos");
      console.log(erro);
  });
```

Um pouco verboso nosso código? Podemos enxugá-lo. No lugar de recebermos uma *promise*, podemos passar para a função query duas funções: a primeira, um callback de sucesso; e a segunda, um callback de erro. Nosso controller ficará assim:

```
angular.module('contatooh').controller('ContatosController',
    function($resource, $scope) {

      $scope.contatos = [];

      $scope.filtro = '';
```

```
    var Contato = $resource('/contatos');

    Contato.query(
      function(contatos) {
        $scope.contatos = contatos;
      },
      function(erro) {
        console.log("Não foi possível obter a lista de
                     contatos");
        console.log(erro);
      }
    );
});
```

Deixando mais clara nossa intenção

Podemos deixar ainda mais clara nossa intenção de buscar os contatos criando a função buscaContato e movendo para dentro dela a chamada de Contato.query():

```
angular.module('contatooh').controller('ContatosController',
  function($resource, $scope) {

    $scope.contatos = [];

    $scope.filtro = '';

    var Contato = $resource('/contatos');

    function buscaContatos() {
      Contato.query(
        function(contatos) {
          $scope.contatos = contatos;
        },
        function(erro) {
          console.log("Não foi possível obter a lista de
                       contatos");
          console.log(erro);
        }
```

```
        );
    }
    buscaContatos();
});
```

Du ◻◻o◻h◻Z◻◻◻◻u v u◻h◻Zo◻◻Z ◻◻h◻Z◻◻◻l e◻o◻ h◻◻◻ ◻◻◻◻u ◻

Imagine que você tenha várias funções para serem chamada na inicialização de seu controller. Neste caso, você poderia criar a função scope.init (ou qualquer outro nome) e nela chamar todas as funções de que precisa na inicialização, deixando seu código mais claro:

```
$scope.init = function() {
    buscaContatos();
    tarefa1();
    tarefa2();
};
$scope.init();
```

Em um primeiro momento, $resource não parece trazer grandes vantagens, mas elas aparecerão à medida que formos realizando outras operações.

Aliás, precisamos implementar a funcionalidade de remoção de contatos de nossa lista. Primeiro, precisaremos criar a rota de exclusão em nosso servidor Express, para depois a integrarmos com o AngularJS.

5.7 Ao◻h◻◻◻Z◻o◻ ◻◻◻◻Z ou u h◻◻◻e◻Z◻ E◻◻u ◻◻

Vimos nos capítulos anteriores que estamos utilizando o padrão REST. Tanto isso é verdade que utilizamos o serviço $resource do AngularJS especializado neste padrão.

Precisamos criar uma rota para exclusão de contatos. Um prerrequisito é que ela siga o padrão REST e deve ser capaz de receber um identificador para o contato que desejamos remover da lista.

Configurando a rota

Se pensarmos em REST, o verbo HTTP indicado para nossa ação é DELETE. Resolvemos isso facilmente com o Express, porque, além de ele utilizar a função app.get para rotas com o verbo GET, ele também possui app.post, app.put e até app.delete.

Vamos adicioná-la em nosso arquivo de rotas referente ao recurso contato:

```
// contatooh/app/routes/contato.js

module.exports = function (app) {
    var controller = app.controllers.contato;

    /* rotas anteriores */

    app.delete ('/contatos', controller.removeContato);
};
```

Criamos uma rota para o verbo DELETE utilizando o identificador /contatos. Também já indicamos qual função chamaremos e que ainda precisa ser criada no controller de contatos. Porém, antes de a criarmos, precisamos resolver um problema.

Parametrizando nossa rota

Em algum momento, nossa aplicação utilizando AngularJS precisará enviar o ID do contato selecionado da lista para que ele seja removido. Como capturar esse ID?

Basta alterar um pouquinho o identificador de nosso recurso. Desta forma:

```
app.delete( '/contatos/:id', controller.removeContato);
```

Vimos em um dos capítulos anteriores que o identificador pode trabalhar com um curinga. Em nosso caso, escolhemos o nome :id. O identificador /contatos/ é constante, o que muda é o código do recurso que queremos apagar:

Vejamos um exemplo:

http://localhost:3000/contatos/1

O mesmo endereço para o recurso, dependendo do verbo, gera ações diferentes. Se for GET, retorna o contato identificado pelo ID 1; se o verbo for DELETE, apagará o contato.

Obtendo o ID do contato no controller

Muito bem, já entendemos como configurar nossa rota de exclusão, porém, como acessaremos o ID do contato passado pela URL dentro nosso controller? Já vimos isso na seção 2.12.

Primeiro, vamos criar a função removeContato em nosso controller.

```
// contatooh/app/controllers/contato.js

/* outras funções */

controller.removeContato = function(req, res) {
  console.log('API: removeContato: ');
};
```

Existe a propriedade req.params. Não custa lembrar que nela temos acesso aos parâmetros enviados na requisição. Como utilizamos na configuração de nossa rota o curinga :id, podemos acessar o ID do contato enviado através de req.params.id:

```
// contatooh/app/controllers/contato.js

/* outras funções */

controller.removeContato = function(req, res) {

  var idContato = req.params.id;

  console.log('API: removeContato: ' + idContato);
};
```

Agora basta implementar a lógica que busca o contato pelo seu ID, removendo-o, caso exista:

```
// contatooh/app/controllers/contato.js
controller.removeContato = function(req, res) {
  var idContato = req.params.id;
  contatos = contatos.filter(function(contato) {
      return contato._id != idContato;
  });
  res.send(204).end();
};
```

Repare na chamada à função res.send. Ela recebe como parâmetro o status code 204 (*No Content*) da operação. Em nosso caso, indicamos que a operação correu normalmente.

Agora que já temos nossa rota de exclusão devidamente configurada, vamos implementar no lado do cliente a chamada desta rota através do AngularJS.

5.8 Express: organizando melhor nossas rotas

Criamos no Express nossas rotas, porém você já deve ter reparado que tanto a rota de listagem quanto a de inclusão utilizam o mesmo identificador, mudando apenas o verbo; a mesma coisa ocorre com as rotas de pesquisa e exclusão:

```
// app/routes/contato.js
module.exports = function (app) {
  var controller = app.controllers.contato;

  app.get('/contatos', controller.listaTodos);
  app.post('/contatos', controller.salvaContato);

  app.get('/contatos/:id', controller.obtemContato);
  app.delete('/contatos/:id', controller.removeContato);
};
```

Poderíamos até isolar os identificadores que se repetem em uma variável para depois reaproveitá-los, porém o Express, a partir da sua versão 4, trouxe um novo sistema de rotas com recursos interessantes.

Uma das maneiras de interagirmos com o sistema de rotas é por meio de instância do Express através da função **route**, passando como parâmetro o identificador da rota:

```
// app/routes/contato.js
module.exports = function (app) {
  var controller = app.controllers.contato;

  app.route('/contatos');
  app.route('/contatos/:id');
};
```

Repare que agora não temos mais identificadores duplicados, mas sabemos que cada um deles responderá a mais de um verbo HTTP. É por isso que podemos encadear a chamada às funções get, post, delete e put:

```
module.exports = function (app) {
  var controller = app.controllers.contato;

  app.route('/contatos')
    .get(controller.listaTodos)
    .post(controller.salvaContato);

  app.route('/contatos/:id')
    .get(controller.obtemContato)
    .delete(controller.removeContato);
};
```

Para um mesmo identificador, podemos associar diferentes verbos HTTPs, tornando a declaração de nossas rotas mais enxuta e mais fácil de manter.

5.9 Ru [illegible]

Agora que adicionamos no lado do servidor a rota para exclusão de contatos, precisamos implementar no lado do cliente sua chamada através do AngularJS.

A diretiva ng-click mais uma vez

Nosso primeiro passo será adicionar a diretiva ng-click no botão Remover de nossa parcial contatos.html:

```
<!-- public/partials/contatos.html -->
...
  <button
    ng-click=
```

O problema é que sua chamada está muito genérica. Qual contato queremos remover? É por isso que ela recebe um objeto como parâmetro e, nele, indicamos o nome do parâmetro que será enviado na requisição e seu valor:

```
$scope.remove = function(contato) {
    Contato.delete({id: contato._id});
};
```

Estamos dizendo para que o serviço envie no parâmetro id o valor de contato._id, isto é, o ID do contato selecionado. Como nosso servidor está preparado para extrair essa informação, tudo deveria funcionar, porém, precisamos realizar uma nova consulta logo após a exclusão do contato, para que nossa lista seja atualizada. Até poderíamos remover o item da lista no lado do cliente, mas optaremos pela primeira opção.

Lembra das *promises*? Pois é, a função delete pode retornar uma *promise*. A partir daí, já sabemos o que fazer:

```
$scope.remove = function(contato) {
    var promise = Contato.delete({id: contato._id}).$promise;
    promise
    .then(buscaContatos)
    .catch(function(erro) {
      console.log("Não foi possível remover o contato");
      console.log(erro);
    });
};
```

Repare que, para a função then, passamos a função buscaContatos, que será chamada apenas quando a requisição for bem-sucedida. Por fim, na função catch tratamos eventuais erros que possam acontecer durante a requisição.

Há um açúcar sintático para a função delete. Além do primeiro parâmetro, que indica os parâmetros da requisição, podemos passar mais dois: o primeiro, um callback de sucesso; o segundo, um callback de falha:

```
$scope.remove = function(contato) {
  Contato.delete({id: contato._id},
    buscaContatos,
```

```
        function(erro) {
          console.log('Não foi possível remover o contato');
          console.log(erro);
        }
      );
};
```

Por baixo dos panos, a *promise* continua sendo utilizada, porém o código é menos verboso.

Nosso controller final ficará assim:

```
angular.module('contatooh').controller('ContatosController',
    function($resource, $scope) {
      $scope.contatos = [];

      $scope.filtro = '';

      var Contato = $resource('/contatos/:id');

      function buscaContatos() {
        Contato.query(
          function(contatos) {
            $scope.contatos = contatos;
          },
          function(erro) {
            console.log(erro)
          }
        );
      }
      buscaContatos();

      $scope.remove = function(contato) {
        Contato.delete({id: contato._id},
          buscaContatos,
          function(erro) {
            console.log('Não foi possível remover o contato');
            console.log(erro);
          }
        );
```

```
        };
});
```

Já podemos testar a nova funcionalidade. Experimente remover um ou outro contato. Sempre que você apagar a lista inteira, basta parar e reiniciar o servidor para que a lista volte.

5.10 A□ o□u □ □□ □□-□□ou u □□-□□□□

Podemos exibir para o usuário a mensagem "Não há contatos cadastrados" quando a lista estiver vazia. Não podemos simplesmente adicionar este texto em nossa página, pois ele seria sempre exibido. É ai que entra a diretiva **ng-hide**. Elementos com esta diretiva □cam invisíveis quando o valor de ng-hide for true.

Na parcial contatos.html, imediatamente após o parágrafo que exibe o total de usuários cadastrados, vamos adicionar nosso aviso através de outro parágrafo com a diretiva ng-hide. Ela consultará o tamanho do array de contatos. Em JavaScript, qualquer número diferente de zero é considerado true. Por □m, adicionaremos a classe text-info do Bootstrap para dar um apelo visual:

```
<p class="text-info" ng-hide="contatos.length">
    Não há contatos cadastrados
</p>
```

Teste o resultado. Você verá que o parágrafo é exibido apenas quando a lista de contatos estiver vazia, mas o problema é que o parágrafo anterior também é exibido. A ideia é que ele faça o contrário do nosso parágrafo: ele só deve ser exibido se houver elementos na lista. Para isso, existe a diretiva **ng-show**. O elemento só será exibido se o seu valor for true:

```
<p ng-show="contatos.length">
    Contatos cadastrados: {{contatos.length}}
</p>
<p class="text-info" ng-hide="contatos.length">
    Não há contatos cadastrados
</p>
```

Teste mais uma vez, experimente remover todos os itens da lista. Tudo deve continuar funcionando como esperado.

Exibindo avisos para o usuário

Até agora, tratamos possíveis erros que podem acontecer quando acessamos nosso back-end através do AngularJS, inclusive logamos essas informações. O problema é que o usuário □cará sem um feedback sobre o que aconteceu. Que tal usarmos a mesma estratégia que □zemos com os parágrafos para exibir mensagens amigáveis para o usuário?

Vamos adicionar uma nova propriedade no escopo do controller:

```
$scope.mensagem= {texto: ''};
```

Em contatos.html, adicionamos o parágrafo imediatamente após a div com a classe jumbotron:

```
<!-- jumbotron vem antes -->
<p ng-show="mensagem.texto">
    {{mensagem.texto}}
</p>
```

A ideia é simples. Quando houver um erro, preenchemos o texto do objeto mensagem. Se tudo correr bem, limpamos seu valor. Usaremos a diretiva ng-show para exibir a mensagem, apenas se ela existir, pois undefined é avaliado como false em JavaScript:

Alterando nosso controller:

```
angular.module('contatooh').controller('ContatosController',
    function($resource, $scope) {
        $scope.contatos = [];

        $scope.filtro = '';

        $scope.mensagem= {texto: ''};

        var Contato = $resource('/contatos/:id');
```

```
    function buscaContatos() {
      Contato.query(
        function(contatos) {
          $scope.contatos = contatos;
          $scope.mensagem= {};
        },
        function(erro) {
          console.log(erro);
          $scope.mensagem= {
              texto: 'Não foi possível obter a lista'
          };
        }
      );
    }
    buscaContatos();

    $scope.remove = function(contato) {
      Contato.delete({id: contato._id},
        buscaContatos,
        function(erro) {
          $scope.mensagem= {
              texto: 'Não foi possível remover o contato'
          };
          console.log(erro);
        }
      );
    };
});
```

Agora que já sabemos listar e remover contatos, e mostrar mensagens personalizadas para o usuário, precisamos adicionar mais uma funcionalidade em nosso projeto: exibir o contato selecionado da lista.

5.11 Exibindo um contato

Vamos implementar a funcionalidade de exibição do contato selecionado na lista. Para isso, precisamos primeiro incrementar nossa página contato.html, que só exibe o título da página. Utilizaremos mais uma vez

o Bootstrap para darmos um “look” pro□ssional com pouquíssimo esforço:

```
<!-- partials/contato.html -->

<h1 class="text-center">Contato</h1>
<p ng-show="mensagem.texto">
    {{mensagem.texto}}
</p>
<form>
    <div class="form-group">
        <label for="nome">Nome completo</label>
        <input type="text" class="form-control" id="nome"
        name="nome"required ng-model="contato.nome">
    </div>

    <div class="form-group">
        <label for="email">Email</label>
        <div class="input-group">
            <span class="input-group-addon">@</span>
            <input type="email" class="form-control" id="email"
            placeholder="email@exemplo.com" name="email"
            ng-model="contato.email" required>
        </div>
    </div>
    <div>
        <button type="submit" class="btn btn-primary">
            Salvar
        </button>
        <a href="#/" class="btn btn-default">Voltar</a>
    </div>
</form>
```

```
// public/js/main.js
// ...
$routeProvider.when('/contato/:contatoId', {
  templateUrl: 'partials/contato.html',
  controller: 'ContatoController'
});
// ...
```

R□□ Z□ o□ A□□□Z□JS

Repare que, no lado do cliente, usamos a rota /contato/:contatoId no singular para acessarmos um contato especí□co, saindo do padrão REST. Não se preocupe, pois estas rotas são processadas no lado do cliente e a diferença de nomes entre /contatos e /contato nos ajudará com o sistema de rotas do AngularJS.

Nossa rota local trabalha com um curinga. Este curinga, quando passado para a rota /contato, a torna uma rota válida. Repare sua associaç ão com ContatoController:

```
// public/js/controllers/ContatoController.js
angular.module('contatooh').controller('ContatoController',
    function($scope, $routeParams) {
        console.log($routeParams.contatoId);
});
```

Em nosso controller, temos acesso ao ID do contato passado na URL processada pelo sistema de rotas do AngularJS. Agora precisamos que essa URL seja montada dinamicamente toda vez que o usuário clicar sobre o nome do contato da lista.

Vamos alterar a página principal contatos.html, adicionando a diretiva ng-hef na âncora que contém o nome do contato.

```
<!-- public/partials/contatos.html -->
<a ng-href="#/contato/{{contato._id}}">{{contato.nome}}</a>
```

Repare que o valor da diretiva é composto por uma Angular Expression que adiciona o ID do contato processado pela diretiva ng-repeat. Um ponto importante é que a URL começa com #, sinalizando que aquela URL deve ser processada no lado do cliente, evitando que ela seja submetida para o servidor.

ng-href

Usamos a diretiva **ng-href** no lugar do atributo href para garantir que o link só funcione depois de o AngularJS ter sido carregado e depois de ele ter avaliado a Angular Expression associada à diretiva. Nesse ínterim, qualquer clique no link é inofensivo.

Agora que já conseguimos acessar qualquer contato de nossa lista, disponibilizando seu ID para ContatoController, precisamos buscá-lo lá em nosso servidor para que possamos popular os dados do formulário com suas informações.

Utilizaremos novamente $resource, com a diferença de que chamaremos a função get para buscar o contato pelo seu ID. A ideia aqui é traduzir a rota do AngularJS para uma operação no lado do servidor:

```
// public/js/controllers/ContatoController.js

// injetamos $resource
angular.module('contatooh').controller('ContatoController',
    function($scope, $routeParams, $resource) {

        // aqui continua no plural, é a rota no lado do servidor
        var Contato = $resource('/contatos/:id');

        Contato.get({id: $routeParams.contatoId},
            function(contato) {
                $scope.contato = contato;
            },
            function(erro) {
                $scope.mensagem= {
```

```
                    texto: 'Não foi possível obter o contato.'
                };
                console.log(erro);
            }
        );
});
```

Já podemos testar selecionando um contato da lista. Em nosso servidor Express, temos a rota /contatos/:contatoId que espera receber o ID do contato como parâmetro. Como estamos utilizando $resource, a função get é inteligente o suficiente para montar a URL para nosso REST Endpoint!

Como já havíamos adicionados as AEs na página contato.html no início do capítulo, o AngularJS saberá exibir nossos dados.

Lidando com novos contatos

Já aprendemos a buscar contatos selecionados da lista, mas se você lembrar do nosso layout, nossa parcial contatos.html possui o botão "Novo Contato", que em teoria deveria abrir a parcial contato.html com o formulário em branco para que possamos adicionar um novo contato.

O problema é que a rota para a parcial que exibe o contato só exibirá a página se algum ID for passado, e no caso de um novo contato ainda não temos ID! Se omitirmos o ID, o sistema de rotas do AngularJS não detectará a rota como válida e nos direcionará para a parcial contatos.html.

Podemos resolver esse problema adicionando uma nova rota em main.js também para a parcial contato.html, com a diferença de que ela não recebe parâmetros:

```
// public/js/main.js
// ...
$routeProvider.when('/contato', {
    templateUrl: 'partials/contato.html',
    controller: 'ContatoController'
});
// ...
```

Agora, se tentarmos acessar a URL (http://localhost:3000/k/contato) nosso controller será carregado e a página exibida, porém não faz sentido

realizarmos uma busca no servidor sem passarmos um ID.

Podemos ajustar isso facilmente adicionando uma cláusula IF em nosso controller:

```
// public/js/controllers/ContatoController.js

angular.module('contatooh').controller('ContatoController',
    function($scope, $resource, $routeParams) {

        var Contato = $resource('/contatos/:id');

        if($routeParams.contatoId) {
            Contato.get({id: $routeParams.contatoId},
                function(contato) {
                    $scope.contato = contato;
                },
                function(erro) {
                    $scope.mensagem= {
                        texto: 'Contato não existe. Novo contato.'
                    };
                }
            );
        } else {
            $scope.contato = {};
        }
});
```

No código anterior, realizaremos a busca apenas se o ID do contato for passado para o sistema de rotas do AngularJS e adicionamos um objeto sem propriedades em $scope.contato. Isso não dará erro em nosso formulário, que depende dos atributos do contato? Não, porque o two-way data binding criará as propriedades dinamicamente no objeto caso elas não existam, através da diretiva ng-model.

Precisamos ajustar nosso botão "Novo Contato". A primeira coisa é torná-lo um link, para que possamos disparar nosso sistema de rotas. Depois, adicionar a diretiva ng-href:

```
<a ng-href="#/contato" class="btn btn-primary">Novo Contato</a>
```

Pronto! Agora que temos tudo no lugar, já podemos implementar nossa lógica que salva o contato.

5.12 SZ□□d□ □h□□ Z□ □

Precisamos implementar a lógica que inclui ou atualiza um contato. Sabemos que através da diretiva ng-click podemos executar alguma lógica em nosso controller, porém, como estamos utilizando componentes do HTML 5 em nossa página, podemos pegar carona em seu sistema da validação.

A diretiva ng-submit

No lugar de usarmos a diretiva ng-click no botão "Salva", utilizaremos a diretiva **ng-submit** no formulário da página, permitindo que o browser processe seus componentes, logo, realizando a validação do formulário:

```
<!-- public/partials/contato.html -->
<form ng-submit="salva()">
```

Em ContatoController.js:

```
$scope.salva = function() {
  // lógica de salvamento
};
```

Agora precisamos implementar a lógica de salvamento do contato. Primeiro, vamos imaginar que você tenha selecionado o contato da lista e deseja atualizá-lo. Você lembra quem retornou esse contato para nós? Sim, nosso serviço Contato, criado através de $resource. Esse contato retornado possui outras características adicionadas dinamicamente pelo serviço Contato que até agora nem sabíamos que existiam. Então, para salvar um contato basta:

```
$scope.salva = function() {
  $scope.contato.$save();
};
```

Esta é uma das vantagens de trabalharmos com $resource. Qualquer objeto retornado através dele é incrementado com uma série de funções

voltadas para persistência, em nosso caso, persistência ligada a *REST Endpoints*.

A função $save gera por debaixo dos panos uma requisição do tipo POSTque envia para http://localhost/contatos os dados do contato. E se quisermos exibir uma mensagem de sucesso? A função retorna uma *promise* e sabemos que ela possui a função then:

```
$scope.salva = function() {
  $scope.contato.$save()
      .then(function() {
          $scope.mensagem= {texto: 'Salvo com sucesso'};
          // limpa o formulário
          $scope.contato = new Contato();
      })
      .catch(function(erro) {
          $scope.mensagem= {texto: 'Não foi possível salvar'};
      });
};
```

Estamos quase lá! O problema é que nossa lógica funciona para alteração, mas e quando for um novo contato? Ele não é criado a partir do serviço $resource, logo, não tem a fun ção $save.

Podemos facilmente resolver isso. Quando estivermos criando um novo contato, no lugar de ele receber como valor {}, ele receberá uma nova instância do serviço Contato:

```
// public/js/controllers/ContatoController.js

// código omitido
$scope.contato = new Contato();
// código omitido
```

Pronto! Agora teremos acesso à função $save. Não custa lembrar que os atributos do contato serão adicionados dinamicamente, através da associação ng-model que, além de ler, é capaz de alterar dados dinamicamente.

Se realizarmos um teste, receberemos uma mensagem amigável dizendo que não foi possível salvar o contato. Abrindo o console do navegador, vemos a mensagem:

failed to load resource: the server responded with a status of 404 (Not Found)

Isso acontece porque não temos uma rota preparada para lidar com o verbo POST em nosso servidor. Precisamos programá-la.

Em nosso exemplo, usaremos POST para inclusão e atualização. O servidor distinguirá entre as duas operações verificando no contato a presença ou não de um ID.

> **E [illegible] PUT**
>
> No mundo REST, atualização está associada ao verbo PUT, mas $resource não dá suporte a este verbo. Até poderíamos implementá-lo em nosso serviço, mas o autor preferiu utilizar POST para as duas operações, em uma tentativa de seguir as convenções do AngularJS.

Vamos voltar para nosso back-end e implementar a rota que está faltando.

5.13 Ao[illegible] ou [illegible]

Precisamos implementar nossa rota que inclui e atualiza um contato. Para isso, é necessário registrar primeiro a rota:

```
// contatooh/app/routes/contato.js
app.post('/contatos', contatoController.salvaContato);
```

Agora precisamos adicionar a lógica no controller do Express. O primeiro passo é adicionar logo no início do controller um contador que já começa em três, que funcionará como ID de novos contatos:

```
// contatooh/app/controllers/contato.js
var ID_CONTATO_INC = 3;
```

Para terminar, precisamos da lógica que adiciona ou altera um contato. De alguma maneira, devemos ter acesso ao contato que foi enviado através da requisição POST. No Express, basta acessarmos req.body para ter acesso ao corpo da mensagem, em nosso caso, o contato com suas propriedades:

```
// contatooh/app/controllers/contato.js

controller.salvaContato = function(req, res) {

    var contato = req.body;
    contato = contato._id ?
        atualiza(contato) :
        adiciona(contato);
    res.json(contato);
};

function adiciona(contatoNovo) {

    contatoNovo._id = ++ID_CONTATO_INC;;
    contatos.push(contatoNovo);
    return contatoNovo;
}

function atualiza(contatoAlterar) {

    contatos = contatos.map(function(contato) {
        if(contato._id == contatoAlterar._id) {
            contato = contatoAlterar;
        }
        return contato;
    });
    return contatoAlterar;
}
```

Repare que tanto a atualização quanto inclusão retornam como resposta o contato. No caso de uma inclusão, ele já vem com o ID preenchido.

Pronto! Basta reiniciar o servidor e tentar incluir e cadastrar alguns contatos.

5.14 Su [illegible]: [illegible] u [illegible] h[illegible]

Imagine se em algum momento a URL de nosso *REST Endpoint* muda. Precisaríamos alterá-la em todos os lugares que ela for referenciada. Por mais ín□mo que isso possa parecer, seria interessante se conseguíssemos compartilhar um mesmo código entre nossos controllers e, quando o código mudasse, todos □cassem sabendo dessa alteração.

O AngularJS permite criar serviços reaproveitáveis entre controllers, aliás, é possível que controllers troquem informações através desses serviços.

> **Su [illegible]**
>
> Pode parecer um pouco estranho para alguns um serviço guardar estado, mas no mundo do AngularJS ele pode servir para este propósito.

A ideia aqui é isolarmos os detalhes de criação de nosso `$resource`, já que nossos controllers não precisam conhecer a URL do *REST Endpoint* utilizada. Queremos um baixo acoplamento.

O primeiro passo é criar o arquivo `ContatoService.js`. Utilizaremos o su□xo "Service" para evitar confusão com outros scripts já existentes, o AngularJS não é rígido na escolha de seu nome.

Claro, não podemos esquecer de importá-lo, depois do core do AngularJS e antes da importação dos controllers:

```
<!-- public/au  -->
<script    src="js/services/ContatoService.js"></script>
```

Utilizando a instância de nosso módulo principal, utilizamos a função factory, que recebe como primeiro parâmetro o nome do serviço e, no segundo, a função que o de□ne. Repare que na função temos como dependência `$resource`:

```
// public/js/services/ContatoService.js
```

```
angular.module('contatooh').factory('Contato',
                                        function($resource) {

    return $resource('/contatos/:id');
});
```

Pronto, o que nosso serviço retorna é uma instância de $resource já configurada e pronta para uso. Toda serviço criado com factory deve retornar um objeto.

> **E Z l e u hu**
>
> AngularJS também possui a função service com o mesmo propósito, porém, no lugar de retornar um objeto, ela retorna uma função. Na própria documentação do AngularJS, eles sugerem o uso de factory, pois service entrará em desuso.

Agora, basta trocarmos a injeção de $resource nos controllers ContatoController e ContatosController pelo nome de nosso serviço, Contato. Refatorando nosso código, temos:

```
// public/js/controllers/ContatoController.js

angular.module('contatooh').controller('ContatoController',
    function($scope, $routeParams, Contato) {
      // apagar a linha que constrói o resource
}

// public/js/controllers/ContatosController.js

angular.module('contatooh').controller('ContatosController',
    function(Contato, $scope) {
        // apagar a linha que constrói o resource
}
```

Teste mais uma vez a aplicação. Tudo deve continuar funcionando perfeitamente, pois não alteramos o comportamento do código, apenas sua organização.

Excelente! Ao longo deste capítulo aprendemos a integrar nosso front-end com nosso back-end com AngularJS através dos serviços $http e $resource, este último especializado no consumo de *REST Endpoints*. Vimos também que $resource não faz parte do core do AngularJS e que foi preciso baixar o módulo ngResouce, aliás utilizamos o Bower mais uma vez para esta tarefa.

Onde está a persistência?

Nossa aplicação funciona mas até agora não persistimos dados de verdade. Precisamos integrar nossos REST Endpoints com um banco de dados.

Sabemos que o MongoDB é a tecnologia de persistência da MEAN Stack. Antes de realizarmos essa integração, precisamos conhecer o básico deste banco NoSQL. É o que veremos no próximo capítulo.

CZ£ –¶ 6

MongoDB: banco de dados baseado em documento

"Não é o tamanho do cão na luta, é o tamanho da luta no cão"
– Mark Twain

6.1 Ru Zh Z N SQL: f Z h hu o

Talvez um dos mais bem-sucedidos projetos de so ware em nossa história seja o **banco de dados relacional**. Seus princípios criados em 1970 continuam rmes e presentes em banco de dados comerciais e open sources, constituindo o **paradigma relacional**.

Este tipo de banco possui uma estrutura tabular e relacional baseada fortemente em esquemas e, nele, realizamos consultas e resolvemos seus relacionamentos através de uma linguagem padrão chamada SQL (*Structured*

Query Language), mais notadamente o padrão ANSI 92 adotado por diversos bancos.

Com o tempo, um novo paradigma apareceu: o NoSQL. Esse termo foi cunhado em 1998 pelo desenvolvedor Carlo Strozzi e mais tarde reforçado por Eric Evans para indicar bancos que não seguem o padrão SQL. Longe de ser um termo pejorativo, a ideia do NoSQL é a de bancos que não se baseiam em relacionamentos nem em esquemas, tão prezados nos bancos de dados relacionais. Nele, pela ausência de esquemas, a validação e integridade dos dados são de responsabilidade da aplicação.

Uma questão arquitetural?

Cada paradigma apresentado ataca problemas diferentes e sua escolha não é uma decisão puramente preferencial, mas de necessidade do projeto, tangendo aspectos arquiteturais e mais abstratos. Realizar juízo de valor em cada uma das abordagens sem um contexto propriamente dito provavelmente resultará em debates in□amados. Precisamos de um contexto a partir do qual delimitaremos o que queremos resolver.

Delimitando o contexto: impedância

No contexto de nossa aplicação Contatooh, tanto um banco de dados relacional quanto um NoSQL demandará do desenvolvedor a tradução dos dados retornados do banco para algo manipulável na linguagem de programação escolhida, em nosso caso, JavaScript.

Essa discrepância da estrutura dos dados armazenados no banco de dados e as estruturas de dados em memória é chamada de **impedância** [2] e em algum momento o desenvolvedor precisará resolvê-la. Quanto menor for a impedância, menos trabalho de conversão o desenvolvedor terá entre as diversas camadas que compõem seu sistema.

6.2 B□□hZ□o□ Z □u □□□ □□□u o□□hZ □□□□□□i □: JSON "u i□□u □u "

Em nossa busca pela menor impedância possível, é quase impossível não destacar o JSON, uma estrutura textual bastante semelhante ao objeto JavaScript. Não podia ser diferente, já que o protocolo HTTP transfere apenas texto. Além disso, o formato JSON demanda que as chaves venham sempre entre aspas, diferente de um objeto JavaScript no qual seu uso é opcional.

As características que acabamos de ver reforçam a ideia de impedância e, por mais que o JSON seja semelhante a um objeto JavaScript, ainda precisará haver uma tradução das informações. Até agora usamos os termos de maneira intercambiável porque o AngularJS realiza o *parse* do JSON e de objetos JavaScript para nós de maneira transparente. É como se ele □zesse algo do tipo:

```
var contatoJSON = '{"nome":"nome do contato",
    "email":"email do contato"}';

// transforma JSON em objeto
var contatoObjeto = JSON.parse(contatoJSON);
console.log(contatoObjeto);
    Object {nome: "nome do contato", email: "email do contato" }

// transforma objeto em JSON/texto
contatoJSON = JSON.stringify(contatoObjeto);
    '{"nome":"nome do contato","email":"email do contato"}'
```

Já utilizamos o JavaScript Object Notation (JSON) como estrutura de dados tanto em nosso servidor Node.js quanto em nosso cliente AngularJS. Será que podemos adotar uma estrutura semelhante no banco de dados?

Bancos NoSQL

Existem no mercado bancos que armazenam dados em estruturas idênticas ou semelhantes ao JSON, como o **Apache CouchDB** (http://couchdb.apache.org) , **JasDB** (http://www.oberaso□ware.com) , **NeDB** (Node Embedded Database) (https://github.com/louischatriot/nedb) , **Terrastore** (https://code.google.com/p/terrastore) entre outros. Qual deles escolher?

Que tal o MongoDB?

A boa notícia é que já temos um banco escolhido, basta voltarmos ao acrônimo da MEAN Stack para vermos que o "M" é de **MongoDB** (http://www.mongodb.org/) .

O MongoDB é um banco de dados baseado em documento com alta performance e disponibilidade e de fácil escalabilidade. Porém, essas características têm seu preço: nem todas as propriedades ACID (atomicidade, consistência, isolamento e durabilidade) tão prezadas nos bancos de dados relacionais são implementadas pelo banco.

> **A□□ u □□Z□ □□□ Z□ M□□□□DB**
>
> Qualquer banco que permita gravar e ler uma estrutura de dados semelhante ao JSON é bem-vindo à MEAN Stack, porém, o MongoDB ainda é o mais evangelizado, inclusive por Valeri Karpov, criador do acrônimo e que na presente data da escrita desde livro trabalha na empresa criadora do MongoDB.

Começando do zero

Antes de pensarmos em qualquer integração do nosso back-end com nosso banco, precisamos instalar o MongoDB. Depois que o ambiente estiver pronto, aprenderemos os conceitos como documento, ObjectId, collections e como realizar consultas.

6.3 I□□□ Z□Zl e□ o□ M□□□□DB

Existem muitas maneiras de se instalar o MongoDB. Variam de acordo com seu sistema operacional e podem ser consultadas no endereço http://docs.mongodb.org/manual/installation.

É altamente recomendável utilizar a versão de 64 Bits em produção pelo fato de o banco na versão de 32 Bits ser limitado a um tamanho de no máximo dois gigabytes. A versão utilizada durante a elaboração deste livro foi a 2.6.6 de 64 Bits para MacOS.

Uma forma de verificar se a instalação do MongoDB está funcionando é trabalhando com seu cliente de linha de comando: o *mongo shell*. É o que veremos na seção a seguir.

6.4 Usando o mongo shell

O MongoDB possui o **mongo shell**, um cliente em linha de comando que nos permite interagir com suas instâncias. Para acessá-lo, basta digitarmos no terminal o comando **mongo** seguido de dois parâmetros:

```
$ mongo --port 27017 --host localhost
MongoDB shell version: 2.6.6
connecting to: test
```

Repare que no comando anterior passamos os parâmetros --port e --host com os valores 27017 e localhost, respectivamente. Esses valores já são o padrão do mongo shell justamente por eles também serem o padrão da instância do MongoDB que roda em sua máquina.

Vamos parar o mongo shell com a tecla de atalho CONTROL + C (Windows e MacOS) e chamá-lo apenas com o comando mongo:

```
$ mongo
MongoDB shell version: 2.6.6
connecting to: test
```

Logo de início, é exibida no shell a versão do cliente que estamos usando e que estamos conectados ao banco test. Este é o banco padrão do MongoDB no qual podemos realizar qualquer alteração sem medo.

6.5 Olhando mais de perto o mongo shell

Um dos pontos que torna o MongoDB ainda mais interessante na MEAN Stack é a maneira como trabalha com documentos, conceito fundamental para este banco NoSQL.

Características do documento

Um documento pode ser armazenado em diferentes formatos hierárquicos como XML ou JSON com os dados associados a uma estrutura de chave e valor. É através de uma chave específica do documento que temos acesso ao valor associado. Porém, dos diversos formatos existentes no mercado, o MongoDB armazena seus documentos no formato **BSON** (Binary JSON), muito parecido com JSON. Comparemos a estrutura desses dois formatos:

```
// JSON
{
    "nome" : "Flávio Almeida"
}

// BSON
{
    "_id" : ObjectId( "5303e0649fd139619aeb783e")
    "nome" : "Flávio Almeida"
}
```

A grosso modo, a diferença entre BSON e JSON mora na quantidade dos tipos de dados suportados em cada uma deles: enquanto JSON possui seis tipos (Array, Boolean, null, Number, Object e String) o BSON, possui mais de 15 tipos! Veremos alguns desses tipos, porém há um que merece destaque neste momento: ObjectId.

6.6 O □□□ Of□u h□ Io

Todo documento criado pelo MongoDB recebe automaticamente a chave _id contendo como valor padrão um objeto do tipo ObjectId. Voltando ao exemplo da seção anterior:

```
// BSON
{
    "_id" : ObjectId( "5303e0649fd139619aeb783e")
    "nome" : "Flávio Almeida"
}
```

Se fizermos uma analogia com o banco de dados relacional, _id é uma coluna que representa a chave primária da tabela. Sendo assim, não podemos ter mais de um documento com um mesmo ObjectId, contudo, veremos mais à frente que o MongoDB trabalha com o conceito de collection, algo que lembra tabelas no mundo relacional. Dessa maneira, um mesmo ObjectId pode existir em diferentes collections.

> Quando você instala o MongoDB em sua máquina, pelo menos uma instância você terá rodando. Cada instância pode ter vários bancos de dados e cada banco pode ter muitas collections.

6.7 Criando um novo banco de dados

Estamos conectados ao banco test, mas será que existem outros bancos? É através do comando **show dbs** que listamos todos os bancos da instância do MongoDB a que estamos conectados:

```
> show dbs
test 0.203125GB
```

Por enquanto, só temos o banco test e somos capazes de ver quanto ele ocupa de espaço em disco.

Precisamos agora criar um banco que contenha os dados da nossa aplicação Contatooh. Por conveniência, criaremos um banco de mesmo nome, apenas começando com letra minúscula através do comando use contatooh:

```
> use contatooh
switched to db contatooh
```

Sabemos que o banco ainda não existe, porém, o MongoDB irá criá-lo. O mais interessante é que recebemos a mensagem switched to db contatooh. O mongo shell está nos avisando que o objeto db agora aponta para o banco contatooh. Será verdade? Se imprimirmos o objeto db, veremos que ele realmente aponta para nosso novo banco:

```
> db
contatooh
```

A variável db nos fornece um atalho para o banco em que estamos trabalhando no momento. Se por acaso usássemos o comando use test, nossa variável db apontaria para o banco test.

Agora que já sabemos a criar um banco, aprenderemos como incluir documentos através do mongo shell.

6.8 C□□□u h□ □□□□ u □□u □l e□ ou o□h□u □□ □□

O shell do MongoDB permite criar variáveis nos moldes da linguagem JavaScript. Isso é fantástico, porque se a estrutura de dados BSON do MongoDB é parecida com JSON. Podemos criar um objeto JavaScript com a seguinte estrutura:

```
> var contato = { "nome" : "Nome do Contato"}
```

Inclusive, podemos imprimir a variável no console apenas digitando seu nome:

```
> contato
{ "nome" : "Nome do Contato"}
```

A ideia agora é incluir este objeto em nosso banco contatooh. Não se preocupe se a nossa estrutura ainda não é um BSON, porque o próprio MongoDB resolverá essa impedância no momento da inclusão.

Temos um problema: queremos adicionar um contato a uma tabela chamada contatos, mas sabemos que o MongoDB não trabalha com tabelas e, sim, collections para agrupar documentos. Como criar uma collection para depois incluir nosso contato?

Podemos fazer as duas coisas de uma só vez! Vamos convencionar que o nome da collection que armazenará nossos contatos será contatos. Logo, nossa inclusão □cará assim:

```
> var contato = { "nome" : "Nome do Contato" }
> db.contatos.insert(contato)
WriteResult({ "nInserted" : 1 })
```

Repare que, através do objeto db, acessamos um atributo inexistente chamado contatos. Pela natureza dinâmica de objetos JavaScript, o atributo contatos será adicionado dinamicamente em db e o mais fantástico é que, por convenção, ele será a collection que desejamos criar. Na mesma linha, chamamos a função insert, que recebeu nosso objeto como parâmetro.

C□□Z□o□ h□□□u h□ □□□ □u □Z □□□□u □□Z □ □

Quando adicionamos um documento ao mesmo tempo em que estamos criando uma collection, teremos um atraso no tempo de gravação. Isso só acontece na primeira inclusão, devido ao tempo gasto para criação da collection.

Recebemos como resposta WriteResult({ "nInserted" : 1 }), indicando que acabamos de incluir um documento na collection contatos.

Será que tudo realmente funcionou? Primeiro, vamos listar todos os bancos de nossa instância e veri□car se o banco contatooh foi criado:

```
> show dbs
contatooh       0.203125GB
test    0.203125GB
```

Ótimo, lá está nosso banco contatooh! Agora, precisamos listar suas collections:

```
> show collections
contatos
system.indexes
```

Excelente! Vemos que nosso banco existe e que a collection foi criada. Você também deve ter reparado em system.indexes. Como o próprio nome já diz, ela é uma collection contendo os índices de nosso documento.

6.9 B□□hZ□o□ o□h□□u □□ □□

Já sabemos criar bancos e collections, e até inserir documentos nelas, porém ainda não veri□camos se realmente nosso contato foi gravado. Para isso, pediremos à nossa collection que "ache" nosso documento através da função find:

```
> db.contatos.find();
{
  "_id" : ObjectId("53e1534bf8a1b188b3f276d1"),
  "nome" : "Nome do Contato"
}
```

A execução do comando apresenta uma estrutura de dados semelhante ao contato que incluímos, porém há a presença de uma chave ausente no objeto original: _id.

De onde veio _id mesmo?

Como vimos no início do capítulo, documentos são gravados no formato BSON, muito semelhante ao JSON, mas com suporte maior a tipos e que possui como identi□cador um ObjectId. É justamente um ObjectId o valor da chave _id. Não entraremos nos detalhes de criaç ão utilizados para gerar esse número automaticamente, mas tenha certeza de que nunca teremos ObjectId's duplicados em collections. Dessa maneira, garantimos um identi□cador único para nosso documento.

Descartando banco

Pronto, agora que aprendemos o básico, que tal começarmos novamente do zero para irmos mais longe? Precisamos apagar nosso banco recém criado. Como? Pedimos através do objeto db, que representa nosso banco, o seu descarte através da função db.dropDatabase():

```
> db.dropDatabase()
{ "dropped" : "contatooh", "ok" : 1 }
```

Tu Z hu u Z ou u hx u f Z h hu

Tome muito cuidado na hora em que for descartar um banco. Tenha certeza de que o objeto db realmente aponta para o banco desejado. Esta operação é irreversível.

Agora que descartamos nosso banco, podemos começar novamente com um novinho em folha para continuarmos nosso aprendizado:

```
> use contatooh
switched to db contatooh
```

Criamos pela segunda vez o banco contatooh.

Incluindo vários documentos

Para que mais tarde possamos realizar consultas, precisamos popular nosso banco. Primeiro, criaremos três objetos que representam um contato com nome e email:

```
> var contato1 = { "nome" : "Contato 1 Mongo",
    "email" : "cont1@empresa.com.br" }
> var contato2 = { "nome" : "Contato 2 Mongo",
    "email" : "cont2@empresa.com.br" }
> var contato3 = { "nome" : "Contato 3 Mongo",
    "email" : "cont3@empresa.com.br" }
```

Agora, vamos adicioná-los em nosso banco:

```
> db.contatos.insert(contato1)
WriteResult({ "nInserted" : 1 })
> db.contatos.insert(contato2)
WriteResult({ "nInserted" : 1 })
> db.contatos.insert(contato3)
WriteResult({ "nInserted" : 1 })
```

O que acontecerá se tentarmos adicionar um quarto contato inexistente?

```
> db.contatos.insert(contato4)
ReferenceError: conta4 is not defined
```

Da mesma maneira que em um código JavaScript recebemos uma mensagem de erro, o mongo shell nos dá uma pista do problema. Isso é interessante, principalmente se você faz uso pesado do cliente em linha de comando.

E agora? Como listar nossos contatos? Utilizaremos a mesma função find usada anteriormente:

```
> db.contatos.find()
{
    "_id"    : ObjectId("530b825db61fac75624ccfd1"),
    "nome"   : "Contato 1 Mongo",
    "email" : "cont1@empresa.com.br"
}
{
    "_id"    : ObjectId("530b8260b61fac75624ccfd2"),
    "nome"   : "Contato 2 Mongo",
    "email" : "cont2@empresa.com.br"
}
{
    "_id"    : ObjectId("530b85aab61fac75624ccfd7"),
    "nome"   : "Contato 3 Mongo",
    "email" : "cont3@empresa.com.br"
}
```

Você já deve ter percebido que a função find retorna todos os documentos de nossa collection, porém há mais coisas do que seu olho pode ver!

6.10 O f u h

Aprendemos na seção anterior a listar nossos documentos através da função find. Que tal guardarmos o resultado da fun ção em uma variável para depois imprimirmos seu valor?

```
> var contatos = db.contatos.find()
> contatos
```

```
{
    "_id"    : ObjectId("530b825db61fac75624ccfd1"),
    "nome"   : "Contato 1 Mongo",
    "email"  : "cont1@empresa.com.br"
}
{
    "_id"    : ObjectId("530b8260b61fac75624ccfd2"),
    "nome"   : "Contato 2 Mongo",
    "email"  : "cont2@empresa.com.br"
}
{
    "_id"    : ObjectId("530b85aab61fac75624ccfd7"),
    "nome"   : "Contato 3 Mongo",
    "email"  : "cont3@empresa.com.br"
}
```

Funciona! Nossa variável contatos contém todos nossos contatos. Será mesmo? Vamos imprimi-la novamente:

```
> contatos
```

Desta vez, nada acontece! Por quê? Isso ocorre porque a lista retornada pelo método find da collection não é um array de objetos, mas um **cursor** criado pelo MongoDB, que buscará nossos dados apenas quando precisarmos.

Quando executávamos a função find diretamente ou quando guardávamos seu retorno em uma variável e depois a imprimíamos, o que o shell do MongoDB na verdade fazia era chamar a função next() do cursor, que retornava um contato a cada chamada.

O mongo shell era inteligente chamando essa função várias vezes até que todos os contatos fossem listados. É por isso que, ao tentarmos imprimir novamente o cursor, nada é impresso, porque o cursor já atingiu o final da lista.

Que tal fazermos um teste para verificar se isso é realmente verdade? No lugar de imprimirmos a variável contatos no console, chamaremos manualmente o método next():

```
> var contatos = db.contatos.find()
> contatos.next()
```

```
{
    "_id"    : ObjectId("530b825db61fac75624ccfd1"),
    "nome"   : "Contato 1 Mongo",
    "email"  : "cont1@empresa.com.br"
}
> contatos.next()
{
    "_id"    : ObjectId("530b8260b61fac75624ccfd2"),
    "nome"   : "Contato 2 Mongo",
    "email"  : "cont2@empresa.com.br"
}
> contatos.next()
{
    "_id"    : ObjectId("530b85aab61fac75624ccfd7"),
    "nome"   : "Contato 3 Mongo",
    "email"  : "cont3@empresa.com.br"
}
```

Realmente, para cada chamada de next, o primeiro, o segundo e o terceiro contato são impressos no console. Porém, se chamarmos a função pela quarta vez teremos um erro:

```
> contatos.next()
error hasNext: false at src/mongo/shell/query.js:127
```

Este erro aconteceu porque não há mais elementos no cursor, ou seja, ele já foi totalmente percorrido. É por isso que o cursor possui a função hasNext, que retorna true apenas se existir um próximo elemento.

Que tal agora avançarmos um pouco mais em termos de consulta no MongoDB? É isso o que veremos na próxima seção.

6.11 B□□hZ□o□ h□□h□□□ v□□□

Muitas vezes queremos encontrar apenas um documento, por exemplo, o primeiro que encontrarmos em nossa collection. Para isso existe a fun ção **findOne** presente em toda collection:

```
> db.contatos.findOne()
{
```

```
    "_id"     : ObjectId("530b825db61fac75624ccfd1"),
    "nome"    : "Contato 1 Mongo",
    "email"   : "cont1@empresa.com.br"
}
```

Repare que, dos três contatos que havíamos cadastrados, apenas o primeiro foi retornado. Faremos a mesma coisa novamente, só que dessa vez guardando o resultado em uma variável:

```
> var contato = db.contatos.findOne()
> contato.next()
TypeError: Object [object Object] has no method 'next'
```

Espere um pouco! Aprendemos que a função `find` retorna um cursor que nos permite acessar todos os documentos de nossa `collection` através da função `next`, mas isso parece não funcionar quando usamos `findOne`.

Isso acontece porque a função `findOne` sempre retorna um objeto do banco, não um cursor. Como não há a necessidade de iterar sobre uma lista de documentos, não há necessidade de um cursor. Desta forma, podemos imprimir quantas vezes quisermos a variável `contato`, que ela será sempre exibida. Caso ela fosse um cursor, depois de ter atingido seu último documento, nada mais seria impresso no console:

```
> contato
{
    "_id"     : ObjectId("530b825db61fac75624ccfd1"),
    "nome"    : "Contato 1 Mongo",
    "email"   : "cont1@empresa.com.br"
}
> contato
{
    "_id"     : ObjectId("530b825db61fac75624ccfd1"),
    "nome"    : "Contato 1 Mongo",
    "email"   : "cont1@empresa.com.br"
}
```

Nem sempre queremos listar todos os documentos de nossa `collection` e muitas vezes queremos utilizar critérios na hora da

consulta. Banco de dados relacionais utilizam SQL com esta nalidade, porém o MongoDB possui uma linguagem de consulta própria baseada no tão familiar JSON.

Criando critérios

Critérios são criados através da estrutura de dados JSON e são passados para as funções find e findOne da collection:

```
> var criterio = { "email" : "cont2@empresa.com.br" }
> var contato = db.contatos.find(criterio)
> contato
{
    "_id"   : ObjectId("530b8260b61fac75624ccfd2"),
    "nome"  : "Contato 2 Mongo",
    "email" : "cont2@empresa.com.br"
}
```

No exemplo anterior, criamos um critério que nada mais é do que um JSON com a chave email recebendo o valor que desejamos buscar no banco. O MongoDB é inteligente para comparar a chave email com a mesma chave presente em todos os documentos de nosso banco, retornando apenas aqueles que atenderem o critério passado.

o . o({})

A função find sempre recebe um critério, mesmo quando ele é omitido. Na sua omissão, o MongoDB se encarrega de passar um objeto sem chaves como critério. Sendo assim, as chamadas find() e find({}) retornam o mesmo resultado.

Existe algo parecido com o like do SQL?

E se quisermos encontrar todos os contatos que contenham o valor tato como parte do nome, ignorando maiúscula ou minúscula? Se você é conhecedor de expressão regular, já sabe a resposta. Podemos passar como critério expressões regulares:

```
> var criterio = { "nome" : /tato/i }
> var contatos = db.contatos.find(criterio)
```

Essa característica permite criar consultas versáteis, utilizando todo o poder das expressões regulares.

Quantos contatos cadastrados?

Podemos saber quantos contatos temos cadastrados no banco sem precisarmos retornar um cursor. Isso é feito através da função **count** chamada diretamente na collection pesquisada:

```
> db.contatos.count()
2
```

Inclusive, podemos passar um critério para a função count:

```
> db.contatos.count({ "nome" : /to 2/i })
1
```

Nesse exemplo, contamos quantos possuem como parte do nome a expressão to 2 no modo case insensitive.

Um pouco mais sobre consultas

Não é incomum um contato ter mais de um e-mail:

```
> var contatoComEmails =
    {
      "nome": "Contato com emails",
      "emails": ["contato@email.com.br", "pessoal@empresa.com.br"]
    }
> db.contatos.insert(contatoComEmails)
WriteResult({ "nInserted" : 1 })
```

Muitas vezes queremos encontrar um contato a partir de determinado e-mail:

```
> db.contatos.find({"emails" : "pessoal@empresa.com.br"})
{
```

```
    "_id" : ObjectId("53eb582e7514ce1e85529195"),
    "nome" : "Contato com emails",
    "emails" : [ "contato@email.com.br",
                 "pessoal@empresa.com.br" ]
}
```

Essa consulta buscará o e-mail procurado na lista de e-mails do contato.

6.12 Qu ma u u h

Qualquer critério passado para as funções find e findOne terão suas chaves consideradas como cláusulas AND. Mas se quisermos realizar um OR? Para isso, utilizaremos um dos vários **query selectors** do MongoDB:

```
> db.contatos.find({
    "$or" : [
        {
            "email" : "cont2@empresa.com.br"
        },
        {
            "nome" : "Contato 1 Mongo"
        }
    ])
```

Repare que a função find recebe como parâmetro um objeto com a chave $or, um *query selector*. O valor desta chave é um array com as condições que desejamos incluir. O código anterior listará os contatos que atendam ao nosso critério.

Podemos ainda obter todos os contatos que não contenham como e-mail cont2@empresa.com.br, através do *query selector* $ne:

```
> db.contatos.find({
    "email" : {
        "$ne" : "cont2@empresa.com.br"
    }
});
```

Os *query selectors* $or e $ne são dos grupos *comparison* e *logical*, respectivamente. Esses grupos possuem outros seletores e há outros grupos.

Você pode obter na própria página do MongoDB (http://docs.mongodb.org/manual/reference/operator/query/) uma lista de todos os seletores separados por grupos.

6.13 I□ou □□o□ o□h□u □□ □□

Imagine se tivéssemos 30 milhões de contatos e desse universo quiséssemos encontrar aqueles que possuem determinado e-mail? Em termos de critério, não precisamos fazer nada, pois aprendemos a realizar consultas com critérios. Porém, será que a nossa consulta será rápida o su□ciente?

É comum no mundo relacional a criação de índices para colunas comumente consultadas para aumentar a performance das buscas. Isso também é possível no MongoDB. Podemos indicar que determinada chave do documento é indexada através da função ensureIndex:

```
> db.contatos.ensureIndex({ "email" : 1 })
{
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 1,
      "numIndexesAfter" : 2,
      "ok" : 1
}
```

Repare que a função ensureIndex recebe como parâmetro um JSON com a chave que queremos indexar. O valor 1 indica que queremos um índice ascendente, mas é possível trabalhar na ordem decrescente utilizando -1 como valor.

Podemos ainda listar todos os índice de uma *collection* através da função getIndexes:

```
> db.contatos.getIndexes()
[
      {
            "v" : 1,
            "key" : {
                  "_id" : 1
            },
```

```
        "name" : "_id_",
        "ns" : "contatooh.contatos"
    },
    {
        "v" : 1,
        "key" : {
            "email" : 1
        },
        "name" : "email_1",
        "ns" : "contatooh.contatos"
    }
]
```

Além disso, é possível criar índices únicos. Primeiro, apagaremos o índice que acabamos de criar através da função dropIndex da collection que recebe como parâmetro o nome do índice criado:

```
> db.contatos.dropIndex('email_1')
{ "nIndexesWas" : 3, "ok" : 1 }
```

Agora, criamos novamente, mas desta vez como único:

```
> db.contatos.ensureIndex( { email: 1 }, { unique: true } )
{
    "createdCollectionAutomatically" : true,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

Veja que passamos como segundo parâmetro um JSON com a chave unique com valor true. Existem outras possibilidade para a criaç ão de índices e você pode consultá-las em http://docs.mongodb.org/manual/reference/method/db.collection.ensureIndex/.

6.14 Ru □□□Z□o□ o□h□u □□ □□Z□hZ□

Não é raro querermos trabalhar apenas com algumas chaves do documento. Até agora retornamos sempre o documento inteiro. Podemos modi□car esse

comportamento indicando quais chaves queremos retornar passando um segundo parâmetro para a função find ou findOne:

```
> db.contatos.find({}, { "nome" : 1})
{
    "_id" : ObjectId("530b825db61fac75624ccfd1"),
    "nome" : "Contato 1 Mongo",
}
{
    "_id" : ObjectId("530b8260b61fac75624ccfd2"),
    "nome" : "Contato 2 Mongo",
}
{
    "_id" : ObjectId("530b85aab61fac75624ccfd7"),
    "nome" : "Contato 3 Mongo",
}
```

Utilizamos novamente a função find, que recebe como primeiro parâmetro o critério {} que nos retornará todos os contatos. O segundo parâmetro é a novidade. O parâmetro é um objeto que contém as chaves desejadas e em nosso caso queremos apenas uma chave, o nome. Um olhar atento demonstra que a chave possui o valor 1. Isso é importante, já que indica para o MongoDB que queremos incluir aquela chave na consulta.

Por padrão, os documentos retornados vêm com seus respectivos ObjectIds, mesmo sem termos solicitado essa informaç ão. Podemos resolver isso facilmente indicando quais chaves queremos excluir do documento resultante da consulta. O procedimento é o mesmo que zemos antes, porém, no lugar de 1, usamos o valor 0:

```
> db.contatos.find({}, { "nome" : 1, _id : 0 })
{
    "nome" : "Contato 1 Mongo"
}
{
    "nome" : "Contato 2 Mongo"
}
{
    "nome" : "Contato 3 Mongo"
}
```

Já sabemos pesquisar documentos seguindo alguns critérios, mas nem sempre queremos atualizá-los, muitas vezes queremos removê-los!

6.15 Ru [illegible]

A remoção de documentos no MongoDB é feita através da função remove chamada diretamente na collection alvo:

```
> db.contatos.remove({ "email" : "cont1@empresa.com.br" })
WriteResult({ "nRemoved" : 1 })
```

Repare que criamos o critério da mesma maneira que criamos para nossas consultas. A lógica é a mesma.

Vamos verificar se nosso contato foi realmente removido:

```
> db.contatos.find()
{
    "_id"    : ObjectId("530b8260b61fac75624ccfd2"),
    "nome"   : "Contato 2 Mongo",
    "email" : "cont2@empresa.com.br"
}
{
    "_id"    : ObjectId("530b85aab61fac75624ccfd7"),
    "nome" : "Contato 3 Mongo",
    "email" : "cont3@empresa.com.br"
}
```

C[illegible]

Se quisermos apagar irreversivelmente todos os documentos de nossa collection, basta chamar a função db.remove sem argumentos:

```
> db.contatos.remove()
```

Já aprendemos a adicionar, listar e a remover documento de nossas collections, falta ainda entender como o MongoDB lida com atualizações.

6.16 A□ □Z□□□□o□ o□h□□u □□ □□

Um banco de dados não armazena dados apenas, ele permite que esses dados sejam atualizados de acordo com algum critério. Por exemplo, precisamos atualizar o contato com e-mail cont3@empresa.com.br. Já aprendemos como obter contatos de acordo com um critério.

```
> var criterio = { "email" : "cont3@empresa.com.br" }
> var contato = db.contatos.findOne(criterio);
> contato
{
      "_id"     : ObjectId("530b85aab61fac75624ccfd7"),
      "nome" : "Contato 3 Mongo",
      "email" : "cont3@empresa.com.br"
}
```

Utilizamos a função findOne porque temos interesse em apenas um contato. Aliás, não custa lembrar que o retorno será um objeto que representa nosso contato e não um cursor, pois não utilizamos a função find.

Temos certeza de que temos o objeto que estamos procurando. Será que alterar sua chave nome é su□ciente para sincronizá-lo com o banco?

```
> contato.nome = "Nome Alterado"
```

Agora, vamos buscar novamente do banco, mas sem guardar o resultado em uma variável:

```
> db.contatos.findOne(criterio)
{
      "_id"     : ObjectId("530b85aab61fac75624ccfd7"),
      "nome" : "Contato 3 Mongo",
      "email" : "cont3@empresa.com.br"
}
```

Não funcionou. Isso acontece porque somos nós os responsáveis pela sincronização do nosso objeto com o banco, isto é, precisamos atualizar o documento persistido pelo MongoDB com os dados do novo objeto.

A função update

Para atualizar documentos, utilizamos a função update, também chamada diretamente na collection que contém nosso documento. Ela recebe como primeiro parâmetro o critério do documento procurado e, como segundo, as chaves que desejamos modificar no documento. Em nosso caso, estamos enviando o objeto inteiro com todas as suas chaves:

```
> db.contatos.update(criterio, contato)
WriteResult({ "nMatched" : 1, "nUpserted" : 0 })
```

Será que ele foi realmente atualizado? Vamos listar novamente todos os documentos e verificar a alteração:

```
> db.contatos.find()
{
    "_id"    : ObjectId("530b8260b61fac75624ccfd2"),
    "nome"   : "Contato 2 Mongo",
    "email" : "cont2@empresa.com.br"
}
{
    "_id"    : ObjectId("530b85aab61fac75624ccfd7"),
    "nome" : "Nome Alterado",
    "email" : "cont3@empresa.com.br"
}
```

Quando realizamos nosso update, o console exibiu a mensagem WriteResult({ "nMatched" : 1, "nUpserted" : 0 }), indicando que apenas um documento foi encontrado com o critério passado através da chave nMatched e que nenhum nUpserted foi realizado. Mas o que significa esse nUpserted? É o que veremos a seguir.

Document Replace

Conseguimos sincronizar a modificação realizada em nosso objeto com o documento no banco. Essa estratégia de sobrepor o documento do banco com um novo documento é chamada de **document replace**.

W[illegible] u C[illegible]hu [illegible]: F[illegible]u Z[illegible]o[illegible]u [illegible]. S Z[illegible]u W[illegible] u

Só a partir da versão 2.6 o MongoDB começou a utilizar para escritas o padrão **safe writes**, que noti□ca a aplicação em caso de falhas. Versões anteriores utilizam o padrão fire and forget. Este tipo de escrita não aguarda uma resposta do banco, o que permite que um grande volume de atualizações possam ser enviadas sem que sua aplicação □que bloqueada aguardando uma resposta. Porém, se algum erro ocorrer na conexão entre sua aplicação e seu banco, a aplicação não será noti□cada. Dependendo do contexto, esse cenário de incerteza pode ser inadmissível.

6.17 Ru Z[illegible]o[illegible]u [illegible]

Muitas vezes queremos realizar uma atualização, mas caso o documento não exista, queremos que ele seja incluído. Podemos solicitar esse comportamento passando o valor true como o terceiro argumento da função update:

```
> var contato4 = { "nome" : "Contato 4 mongo",
    "email" : "cont4@empresa.com.br" }
> db.contatos.update({ "nome" : /4/ }, contato4, true);
WriteResult({
    "nMatched" : 0,
    "nUpserted" : 1,
    "nModified" : 0,
    "_id" : ObjectId("53e8e3eaf988309060c64744")
})
```

Para termos certeza de que nosso update funcionou, listaremos mais uma vez nossos contatos:

```
> db.contatos.find()
{
    "_id"    : ObjectId("530b8260b61fac75624ccfd2"),
    "nome"   : "Contato 2 Mongo",
    "email" : "cont2@empresa.com.br"
```

```
}
{
    "_id"    : ObjectId("530b85aab61fac75624ccfd7"),
    "nome"   : "Nome Alterado",
    "email"  : "cont3@empresa.com.br"
}
{   "_id"  : ObjectId("53e8e3eaf988309060c64744"),
    "nome"   : "Contato 4 mongo",
    "email"  : "cont4@empresa.com.br"
}
```

Nosso upsert funcionou, excelente!

Um problema não esperado

Imagine o cenário no qual queremos modificar apenas o e-mail de um contato. Que tal aplicar um document replace? É isso que faremos!

```
> db.contatos.update({ "nome" : /2/ },
    { "email" : "somudei@email.com" })
WriteResult({ "nMatched" : 1, "nUpserted" : 0,
              "nModified" : 1 })
```

Pronto! O shell do MongoDB nos informa que um documento foi modificado, excelente! Verificando a alteração:

```
> db.contatos.find()
{
    "_id" : ObjectId("530b8260b61fac75624ccfd2"),
    "email" : "somudei@email.com"
}
```

Espere um minuto! Não era bem este o resultado desejado. Repare que o documento inteiro foi substituído por um documento que agora contém apenas a chave email, apesar de o ObjectId ainda ser o mesmo. Perdemos informações!

6.18 g□u □ □□o□□hZo□□ ou Z□ □Z□□Zl e□

Podemos atualizar documentos através de *modifiers*, modi□cadores. Sua sintaxe é um pouco mais verbosa, porém ela permite que sejamos precisos na atualização do documento:

```
> db.contatos.update(
      { "email" : "cont4@empresa.com.br"},
      {
        "$set" :
          {
            "nome" :  "Mais uma alteração"
          }
      }
 )
WriteResult({ "nMatched" : 1, "nUpserted" : 0 })
```

Vejamos a sintaxe anterior. Nela, a função update recebe como primeiro parâmetro nosso critério, o que não é novidade para nós. Aprendemos também que o segundo parâmetro é um JSON que contém as chaves com os valores que desejamos modi□car nos documentos que atenderem nosso critério.

Porém, o objeto passado como segundo parâmetro não recebe diretamente a chave que desejamos modi□car. Ela é de□nida como valor de uma chave especial chamada $set.

A chave $set é um modi□cador permitindo que sejamos pontuais na alteração de um documento. No lugar de sobrepor o documento, como na estratégia document replace, ele apenas altera as chaves de□nidas como valor de $set.

Como sempre, veri□caremos se realmente a modi□cação foi pontual listando todos os nossos contatos:

```
> db.contatos.find()
{
     "_id"    : ObjectId("530b8260b61fac75624ccfd2"),
     "email" : "cont2@empresa.com.br"
}
{
     "_id"    : ObjectId("530b85aab61fac75624ccfd7"),
```

```
    "nome"   : "Nome Alterado",
    "email" : "cont3@empresa.com.br"
}
{   "_id"    : ObjectId("53e8e3eaf988309060c64744"),
    "nome"   : "Mais uma alteração",
    "email" : "cont4@empresa.com.br"
}
```

Temos o resultado esperado.

> **O□ou u □□ Z Z □ □Z□□Zl e□□**
>
> O MongoDB não suporta transação, apenas operações atômicas. Estas operações consistem na atualização do documento e outros documentos embutidos. Caso algum erro aconteça, nada será modi□cado no documento. Caso você precise de uma transação que envolva vários documentos, a equipe do MongoDB criou um pattern para "*two phase commits*" disponível em http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits.

6.19 D□h□u □□ □□ u □f□□ □o□□ (E□f u oou o D□h□u □□ □)

Imagine agora que cada contato possua um contato de emergência. Primeiro, buscamos o contato e em seguida sua emergência para no □nal realizarmos a associação:

```
> var contato = db.contatos.findOne({ "email" : /cont2/ })
> var emergencia = db.contatos.findOne({ "email" : /cont3/})
> contato.emergencia = emergencia;
> db.contatos.update({ "_id": contato._id}, contato);
WriteResult({ "nMatched" : 1, "nUpserted" : 0,
              "nModified" : 1 })
```

O primeiro ponto a destacar é que nenhum erro aconteceu. Como o MongoDB não força um esquema, fomos capazes de incluir uma nova informação

ao documento. Os documentos já existentes na collection contatos não terão essa informação.

O segundo ponto, desta vez em forma de pergunta, é o seguinte: como será que o MongoDB gravou a informação? Para sabermos, vamos buscar o contato que acabamos de modi□car:

```
> db.contatos.findOne({ "email" : /cont2/ })
{
      "_id" : ObjectId("530b8260b61fac75624ccfd2"),
      "email" : "cont2@empresa.com.br",
      "emergencia" : {
            "_id" : ObjectId("530b85aab61fac75624ccfd7"),
            "nome" : "Nome Alterado",
            "email" : "cont3@empresa.com.br"
      }
}
```

Nossa busca retornou nosso documento, inclusive o seu contato de emergência! Esse teste demonstra que o MongoDB permite armazenar em chaves outro documento que chamamos de **documentos embutidos** (*embedded documents*).

Uma vantagem desta estratégia é que buscamos apenas um documento sem a necessidade de realizarmos JOINS, como nos bancos de dados relacionais. Uma consequência positiva disso é a rapidez nas consultas.

Há cenários onde trabalhar com Embedded Documents é extremamente viável, por exemplo, uma nota □scal que já vem com todos os seus itens. Porém, no exemplo anterior, podemos ter inconsistência em nossos dados. Vejamos um exemplo:

```
> var emergencia = db.contatos.findOne({ "email" : /cont3/})
> emergencia.nome = "Mais uma vez Alterado"
> db.contatos.update({"_id", emergencia._id}, emergencia)
> db.contatos.findOne({ "email" : /cont2/ })
{
      "_id" : ObjectId("530b8260b61fac75624ccfd2"),
      "email" : "cont2@empresa.com.br",
      "emergencia" : {
            "_id" : ObjectId("530b85aab61fac75624ccfd7"),
```

```
            "nome" : "Nome Alterado",
            "email" : "cont3@empresa.com.br"
        }
}
```

Repare que alteramos diretamente na collection o contato de emergência. Porém, essa modificação não reflete no embedded document: temos um dado inconsistente. Mas será que o uso de embedded documents sempre gerará inconsistência? Com certeza, não.

O exemplo da nota fiscal e seus itens é bastante esclarecedor. Nele, buscamos a nota fiscal com todos seus itens. Nesta relação de composição na qual um item não existe sem uma nota, teremos a certeza de que qualquer alteração no item afetará apenas a nota da qual ele faz parte. Neste caso, a criação da collection item é opcional. No mundo SQL, este mesmo relacionamento envolveria as tabelas nota e item associadas através de um JOIN, o que em teoria pode ser mais lento.

Sendo assim, a pergunta que devemos fazer é: ter um contato de emergência como um embedded document pode nos causar algum problema? Sim, pois nesta relação de agregação não queremos correr o risco de nos basearmos em dados inconsistentes (e-mail desatualizado etc.) na hora em que uma fatalidade acontecer.

Dentro do que acabamos de ver, pode ser que para determinado tipo de aplicação o MongoDB não seja indicado e para outros ele seja altamente recomendado. Porém, em nossa aplicação podemos simular JOINS do mundo SQL no lado da aplicação, o que resolveria o problema da consistência dos dados entre um contato e sua emergência.

6.20 Simulando JOINS no lado da aplicação

O MongoDB não nos oferece uma maneira nativa de criarmos relacionamentos do tipo presente no mundo SQL. Se quisermos realizar algo parecido, a responsabilidade pela criação e resolução desses relacionamentos será de responsabilidade do desenvolvedor.

A maneira mais simples de fazermos isso é, no lugar de usarmos um embedded document, gravarmos apenas seu ObjectId.

```
> var contato = db.contatos.findOne({ "email" : /cont2/ })
> var emergencia = db.contatos.findOne({ "email" : /cont3/})
> contato.emergencia = emergencia._id
> db.contatos.update({ "_id" : contato._id}, contato);
WriteResult({ "nMatched" : 1, "nUpserted" : 0,
                "nModified" : 1 })
```

□□□u □□□ _

Não é raro encontrar programadores que pre□xam a chave que guarda o ObjectId com _. Isso fornece uma pista visual indicando que a chave não guarda um valor qualquer, mas um ObjectId.

Vamos buscar o contato novamente do banco para veri□carmos a mudança:

```
> contato = db.contatos.findOne( "_id", contato._id)
{
    "_id"          : ObjectId("530b8260b61fac75624ccfd2"),
    "email"        : "cont2@empresa.com.br",
    "emergencia" : ObjectId("530b85aab61fac75624ccfd7")
}
```

Agora, precisamos executar uma nova consulta para obter o contato de emergência:

```
> var emergencia =
        db.contatos.findOne({ "_id" : contato.emergencia});
> emergencia
"_id" : ObjectId("530b85aab61fac75624ccfd7"),
    "nome" : "Mais uma vez Alterado",
    "email" : "cont3@empresa.com.br"
```

A solução anterior funciona porque sabemos que o ObjectId armazenado na chave contato.emergencia faz parte da collection contatos, porém um programador que acabou de entrar em nossa equipe saberia com clareza dessa relação? É muito provável que ele procurasse esse

ObjectId em uma collection chamada emergencias, que sequer existe em nosso banco. Será que há alguma forma de fornecermos uma uma pista para o desenvolvedor de qual collection aquele ObjectId faz parte? É isso que veremos a seguir.

6.21 DBRu ⊠⊠:oZ⊠ Zf Z⊠u ⊠u ⊠u ⊠u ⊠hu ⊠

O MongoDB possui um recurso chamado **DBRefs** (*database references*) que nos ajuda a saber de qual collection nossos IDs pertencem. Antes de mais nada, voltaremos para o relacionamento que vimos na seção anterior:

```
> var contato = db.contatos.findOne({ "email" : /cont2/ })
> contato
{
    "_id" : ObjectId("530b8260b61fac75624ccfd2"),
    "email" : "cont2@empresa.com.br",
    "emergencia" : ObjectId("530b85aab61fac75624ccfd7")
}
> var emergencia =
        db.contatos.findOne({ "_id" : contato.emergencia});
```

Na consulta anterior, temos o ObjectId de um contato de□nido diretamente na chave contato.emergencia, porém agora ele será o valor da chave $id de um objeto DBRefs:

```
> contato.emergencia = {
        "$ref" : "contatos",
        "$id" : emergencia._id
}
```

Repare que, além de $id, adicionamos também a propriedade $ref que serve para indicar a collection à qual nosso ObjectId pertence.

> **gof**
>
> É possível adicionar a chave $db para indicar de qual banco a collection de□nida na chave $ref faz parte, porém essa informação não é suportada por todos os drivers.

Agora atualizaremos nosso contato no banco através de um document replace:

```
> db.contatos.update({"_id" : contato._id}, contato);
WriteResult({ "nMatched" : 1, "nUpserted" : 0,
              "nModified" : 1 })
```

Logo em seguida, buscaremos o contato do banco novamente guardando o resultado na mesma variável:

```
> contato = db.contatos.findOne({"_id" : contato._id});
> contato
{
"_id" : ObjectId("530b8260b61fac75624ccfd2"),
"email" : "cont2@empresa.com.br",
"emergencia" : DBRef("contatos",
                     ObjectId("530b85aab61fac75624ccfd7"))
}
>
```

Repare no console que o tipo da chave emergencia é um DBRef. Que tal imprimirmos no console a emergência do contato?

```
> contato.emergencia
DBRef("contatos", ObjectId("530b85aab61fac75624ccfd7"))
```

Onde está nosso contato? Repare que o mongo shell não busca automaticamente para nós nosso contato. A responsabilidade de resolver o relacionamento continua sob responsabilidade da aplicação, contudo alguns drivers utilizam a informação do DBRef para resolver o relacionamento.

Resolvendo relacionamentos DBref

Podemos solucionar este problema no mongo shell criado uma função especializada para resolver relacionamentos. Chamaremos a função de refResolver:

```
> function refResolver(ref) {
  return db[ref.$ref].findOne({"_id": ref.$id})
}
```

```
> var emergencia = refResolver(contato.emergencia);
> emergencia
{
    "_id" : ObjectId( "530b85aab61fac75624ccfd7"),
    "nome" : "Nome Alterado",
    "email" : "cont3@empresa.com.br"
}
```

Por mais que nossa solução funcione, para cada associação de nosso documento será necessário realizar uma consulta em separada para obter cada embedded document associado. Esse comportamento não é tão performático quanto os JOINS SQL que permitem que, em uma única consulta, resolvamos todos os relacionamentos. Esse problema se torna mais eminente quando temos um array com vários ObjectIds como chave de um documento, pois geraremos N consultas para cada um deles além da consulta do documento principal.

As possibilidades de consultas no MongoDB vão muito além do que vimos. Caso o leitor queria se aprofundar, o próprio site do MongoDB documenta todas as possibilidades possíveis (http://docs.mongodb.org/manual/tutorial/query-documents/) .

MongoDB e seu universo

Aprendemos neste capítulo como criar bancos dentro de uma instância do MongoDB e a organizar documentos através de collections. Também aprendemos a realizar consultas e como remover e alterar documentos existentes.

É possível realizar consultas mais complexas, inclusive configurar o MongoDB para trabalhar em um *cluster* e por aí vai. Porém, o que vimos é suficiente para continuarmos com a implementação do nosso projeto Contatooh.

Atualmente nosso back-end com Express trabalha apenas com dados voláteis em memória sem se preocupar em persisti-los. Toda vez que reiniciamos o servidor, perdemos todas as informações cadastradas.

Nossa próxima tarefa será integrar nosso back-end com o MongoDB, garantindo a persistência dos dados da aplicação.

CZ£ –¶ 7

Integrando Express e MongoDB

"A coisa mais indispensável a um homem é reconhecer o uso que deve fazer do seu próprio conhecimento."
– Platão

No capítulo anterior, aprendemos conceitos fundamentais do MongoDB, inclusive interagimos com seu cliente em linha de comando, o mongo shell. Exercitamos uma série de operações para inclusão, alteração, remoção e consulta de documentos, porém ainda falta integrar nosso back-end com o MongoDB.

7.1 M□□□□ D□□□

É preciso que nossa aplicação seja capaz de se conectar e realizar operações em nosso banco e normalmente isso é feito através de um

driver. O MongoDB possui um driver nativo (https://github.com/mongodb/node-mongodb-native) criado especialmente para o Node.js. Sua instalação é feita através do npm como qualquer outro módulo do Node.js:

```
npm install mongodb@1.4 --save
```

Dentro da pasta contatooh, vamos criar o arquivo consulta.js e realizar um teste que busca um contato pelo seu ObjectId com o seguinte código:

```
// contatooh/consulta.js

var MongoClient = require('mongodb').MongoClient;
var ObjectID = require('mongodb').ObjectID;

// ObjectID de algum contato existente
var _idProcurado = new ObjectID('53ee689e89bd201218944bba');

MongoClient.connect('mongodb://127.0.0.1:27017/contatooh',
    function(erro, db) {
        if(erro) throw err;
        db.collection('contatos').findOne({_id : _idProcurado},
            function(erro, contato) {
                if(erro) throw err;
                console.log(contato);
            }
        );
    }
);
```

Agora, pelo terminal, vamos executar nosso código através do Node.js:

```
node consulta
```

Caso você tenha fornecido um ID válido, o contato será impresso no terminal, caso contrário, será exibido null. Antes de entrarmos nos detalhes do driver, vamos re□etir sobre o papel dos esquemas.

7.2 E□□□u □Z □Z□Z□□ Z□

Imagine uma aplicação web na qual o usuário informa seu nome no local destinado à sua data de nascimento. Sabemos que essa informação será persistida e, como o MongoDB não força um esquema, ele aceitará o documento com esta informação inválida. Talvez seu chefe ache engraçado ver em um relatório o nome do usuário no lugar de sua data de nascimento, ou talvez ele não seja tão bem-humorado assim.

Esquema sim, mas na aplicação

Apesar de o MongoDB não forçar esquemas, isso não quer dizer que esquemas não sejam importantes, inclusive a integridade dos dados neste banco NoSQL é de responsabilidade da aplicação, portanto, responsabilidade do desenvolvedor.

Mongo driver é suficiente?

Poderíamos refatorar agora nosso back-end e utilizar o driver nativo do mongo, mas, sendo um driver de baixo nível, ele não traz nenhum açúcar sintático. Você deve ter reparado nos callbacks aninhados, que podem resultar no callback HELL, dependendo da complexidade da sua consulta.

Também somos os responsáveis pela tradução de strings para ObjectId, caso contrário, a função findONe não funcionará. Outra questão que acabamos de ver é a importância de implementarmos esquemas pela aplicação.

Foi por isso que a própria equipe do MongoDB criou uma abstração sobre o driver nativo do MongoDB que facilita enormemente a criação de esquemas pela aplicação, removendo muito ruído sintático do driver nativo. Essa abstração recebeu o nome de **Mongoose**.

7.3 M□□□□□□u Of□u h□ -D□h□□u □□ M□ou □u □

Mongoose é uma biblioteca de ODM (*Object-Document Modeler*) criada pela equipe do MongoDB. Ela é uma camada entorno do driver do MongoDB que gerencia relacionamentos e executa validações, entre outras funcionalidades.

Mongoose funciona da seguinte maneira: no lado da aplicação, criamos **esquemas** que servem como molde para criação dos modelos da aplicação. Criamos **objetos** a partir desses **modelos** com auxílio do Mongoose e toda alteração realizada nesses objetos é persistida no banco através de métodos específicos presentes na instância do modelo ou diretamente no próprio modelo.

> **Lu f Z ou g u hu**
>
> No lado do cliente, usamos $resource para acessar *REST endpoints*. Você deve lembrar que utilizamos o recurso como uma espécie de modelo, mas que carregava código de infraestrutura, algo questionável. Porém, isso tornou conveniente a chamada das funções responsáveis pela sincronização dos dados do modelo com nosso serviço. A ideia agora é fazer algo parecido com Mongoose, mas desta vez ligando nosso serviço com nosso banco. Podemos fazer uma analogia com o padrão *ActiveRecord* (http://www.martinfowler.com/eaaCatalog/activeRecord.html) utilizado pela comunidade Ruby on Rails.

Instalação

Instalaremos o Mongoose através do npm:

```
npm install mongoose@3.8 --save
```

> O Mongoose já tem como dependência o driver do MongoDB, porém manteremos o driver que baixamos no início desta seção. Ele será utilizado na seção 12.6.

Nosso próximo passo será preparar a conexão com o banco através do Mongoose.

7.4 Gu u hZ o Z h u e

Nossa primeira tarefa será isolar o código de inicialização da conexão em seu próprio arquivo. Criaremos o módulo database.js dentro da pasta config, que terá como dependência o Mongoose. Ele receberá como parâmetro a URL do banco de nossa aplicação que passaremos para o Mongoose, este último responsável pela abertura e gerenciamento da conexão:

```
// config/database.js
var mongoose = require('mongoose');

module.exports = function(uri) {

    mongoose.connect(uri);
}
```

Mesmo sem termos terminado nosso módulo você deve estar se perguntando quem o chamará. A ideia é que nossa conexão seja iniciada com o servidor. É por isso que chamaremos nosso módulo database.js através de nosso server.js:

```
// app/server.js

var http = require('http');
var app = require('./config/express')();
require('./config/database.js')('mongodb://localhost/contatooh');

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

Poderíamos até subir nossa aplicação, porém não teríamos nenhuma informação sobre o estado da conexão. Podemos resolver este problema através dos eventos connected, disconnected e error disparados pela conexão. Basta passarmos um callback para cada um deles com as informações que queremos exibir no console:

```
// config/database.js
```

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

module.exports = function(uri) {

    mongoose.connect(uri);

    mongoose.connection.on('connected', function() {
      console.log('Mongoose! Conectado em ' + uri);
    });

    mongoose.connection.on('disconnected', function() {
      console.log('Mongoose! Desconectado de ' + uri);
    });

    mongoose.connection.on('error', function(erro) {
      console.log('Mongoose! Erro na conexão: ' + erro);
    });
}
```

h□□□u h□ □. h□u Z□ u O□□□u h□ □□□

A função connect cria uma conexão padrão acessível pelos models criados pelo Mongoose, assunto que veremos mais à frente. Para nossa aplicação, isso será su□ciente, porém há aplicações que acessam mais de um banco. Para isso, existe a função createConnection, que retorna a conexão com o banco:

```
var connection = mongoose.createConnection(uri);
```

Entretanto, quando criamos conexões dessa forma, somos os responsáveis em associá-las aos modelos criados pelo Mongoose. Outro ponto é que você terá que garantir que ninguém criará duas conexões para o mesmo banco:

```
var mongoose= require('mongoose');
var connection;
module.exports = function(uri) {
    if(!connection) {
        connection = mongoose.createConnection(uri);
    }
    return connection;
}
```

Nesse exemplo, a conexão será criada apenas na primeira chamada. Qualquer chamada posterior retornará a conexão já criada.

Precisamos garantir que a conexão seja fechada quando nossa aplicação for terminada. Vamos interagir com o objeto process globalmente disponível pelo Node.js e acessível em qualquer local de nossa aplicação.

O objeto process possui o evento SIGINIT disparado quando nossa aplicação é terminada (por exemplo: CONTROL + C no terminal). É através do callback associado a este evento que pediremos ao Mongoose que feche nossa conexão através da função close:

```
// config/database.js
```

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

module.exports = function(uri) {

    mongoose.connect(uri);

    // código anterior omitido

    process.on('SIGINT', function() {
        mongoose.connection.close(function() {
        console.log('Mongoose! Desconectado pelo término da
                    aplicação');
        // 0 indica que a finalização ocorreu sem erros
        process.exit(0);
      });
    });
}
```

Subindo nosso servidor mais uma vez:

```
$ node server
Express Server escutando na porta 3000
Mongoose! Conectado em mongodb://localhost/contatooh
```

Repare no console que o callback associado ao evento connected foi chamado e nossa mensagem exibida. Podemos testar se o callback para o evento disconnected também será chamado finalizando nossa aplicação através de CONTROL + C:

```
$ node server
Mongoose! Conectado em mongodb://localhost/contatooh
Mongoose! Desconectado
Mongoose! Mongoose! Desconectado pelo término da aplicação
```

P□□□ ou h□□u □u □

A função mongoose.connect cria por padrão um pool com cinco conexões, porém esta quantidade pode não ser adequada para todas as aplicações. Podemos alterar esta quantidade passando uma con□guração extra para a função:

```
mongoose.connect(uri, { server: { poolSize: 15 }});
```

Excelente! Já conseguimos nos conectar ao MongoDB através do Mongoose. Agora precisamos aprender como criar esquemas e como utilizá-los para construirmos modelos que serão utilizados pela nossa aplicação.

7.5 C□Z□o□ u □□u □Z□

Vimos que o MongoDB é um banco sem esquemas, não exigindo de seus documentos determinada estrutura. Contudo, isso não torna o esquema menos importante. É por isso que o Mongoose possui o objeto Schema, que de□ne a estrutura de qualquer documento que será armazenado em uma collection do MongoDB. Ele permite de□nir tipos e validar dados.

Nosso primeiro passo será criar o módulo contato.js na pasta app/models. Nele, declararemos um objeto Schema através da função mongoose.Schema que recebe como parâmetro uma série de critérios. Por enquanto, passaremos um objeto vazio como critério:

```
// app/models/contato.js

var mongoose= require('mongoose');

module.exports = function() {

  var schema = mongoose.Schema({});
};
```

Nosso primeiro critério será garantir que as chaves nome e email aceitem apenas strings:

```
// app/models/contato.js

var mongoose= require('mongoose');

module.exports = function() {

  var schema = mongoose.Schema({
    nome: {
      type: String
    },
    email: {
      type: String
    }
  });

};
```

Repare que adicionamos no objeto passado como parâmetro para a função mongoose.Schema as chaves que correspondem às que queremos em nosso documento, porém cada uma delas possui como valor um **objeto de configuração**.

No objeto de configuração, indicamos o Schema Type aceito por cada chave de nosso documento através da chave **type**. Tanto o nome quanto o email devem aceitar apenas texto, por isso atribuímos a type o valor String. Existem outros tipos como Number, Date, Boolean e Array, mas por enquanto o tipo String já é suficiente.

E□□□u □Z u Z□□Z□

Qualquer tipo de esquema do Mongoose pode ser considerado um array, basta envolvê-lo entre colchetes:

```
mongoose.Schema({
  emails: {
    type: [String]
  }
});
```

Dessa forma, garantiremos que apenas arrays contendo o tipo String serão permitidos em nosso documento.

Queremos também que o nome e o email sejam obrigatórios. Isso é feito adicionando a chave required no objeto de con□guração:

```
// app/models/contato.js

var mongoose = require('mongoose');

module.exports = function() {

  var schema = mongoose.Schema({
    nome: {
      type: String,
      required: true
    },
    email: {
      type: String,
      required: true
    }
  });
};
```

Por □m, não pode haver contatos com o mesmo e-mail. Garantimos isso adicionando mais uma chave de con□guração, desta vez a index:

```
// app/models/contato.js

var mongoose = require('mongoose');

module.exports = function() {

  var schema = mongoose.Schema({
    nome: {
      type: String,
      required: true
    },
    email: {
      type: String,
      required: true,
      index: {
        unique: true
      }
    }
  });
};
```

Essas regras já são suficientes para nosso esquema. Porém, para que possamos interagir com o banco, não trabalharemos com o esquema, mas com um modelo criado a partir dele.

Model

Um `Model` é um objeto que corresponde a uma `collection` de nosso banco e utiliza o `Schema` usado em sua criação para validar qualquer documento que tenhamos na `collection`.

É por isso que a última linha do nosso módulo retornará um `Model` criado a partir do nosso `Schema`:

```
// app/models/contato.js

var mongoose= require('mongoose');

module.exports = function() {
```

```
    // código omitido

    return mongoose.model('Contato', schema);
};
```

Pronto! A questão agora é saber como disponibilizaremos o Model criado pelo nosso módulo para o restante da aplicação.

express-load novamente

Na seção 2.9, utilizamos o express-load para disponibilizar todos os controllers da pasta app/controllers diretamente na instância do Express, permitindo que nossas rotas (também carregadas pelo express-load) tivessem acesso aos controllers sem a necessidade da função require. Se abrirmos o arquivo config/express.js, veremos que já havíamos incluído a pasta models:

```
// config/express.js

// código omitido

load('models', {cwd: 'app'})
        .then('controllers')
        .then('routes')
        .into(app);

// código omitido
```

Como a pasta models foi carregada antes da controllers, podemos acessar o model exportado diretamente no controller seguindo a convenção que já aprendemos. Em nosso caso, o padrão de acesso ao model carregado pelo express-load será app.models.contato.

Podemos realizar um teste para ver se tudo está funcionando. Primeiro, apagaremos o banco contatooh através do mongo shell:

```
$ mongo
MongoDB shell version: 2.6.6
connecting to: test
```

```
> use contatooh
switched to db contatooh
> db.dropDatabase()
{ "dropped" : "contatooh", "ok" : 1 }
```

Agora subiremos nossa aplicação:

```
$ node server
Express Server escutando na porta 3000
Mongoose! Conectado em mongodb://localhost/contatooh
```

Note que estamos conectados ao banco contatooh que acabamos de apagar! O Mongoose extraiu o nome do banco da URL de conexão criando-o para nós, inclusive criou a collection contatos, adotando como padrão o nome passado como primeiro parâmetro para a função mongoose.model, porém em *lowercase* e no plural. Para ficar ainda mais interessante, o índice de unicidade para a chave email também foi criado. Vejamos tudo isso através do mongo shell:

```
$ mongo
MongoDB shell version: 2.6.6
connecting to: test

> show dbs
admin (empty)
contatooh 0.078GB
local 0.078GB
test 0.203GB

> use contatooh
switched to db contatooh

> show collections
contatos
system.indexes

> db.contatos.getIndexes()
    // exibe detalhes do índice criado
```

[illegible]u .ou f[illegible]

Você pode visualizar no terminal todos os comandos executados pelo Mongoose habilitando seu *debbuger*:

```
// config/database.js

mongoose.set('debug',true);
```

7.6 U[illegible]o[illegible]ou [illegible]

Aprendemos na seção anterior a criar um Model a partir de um Schema. Disponbilizamos o Model para nosso controller através do express-load utilizando a convenção de acesso app.models.contato. Agora, precisamos reescrever o código de nosso controller.

Vamos editar app/controllers/contato.js removendo a implementação de todas as funções do controller e apagando a lista [illegible]xa de contatos e a variável de autoincremento que criamos. No [illegible]nal, nosso controller deve [illegible]car assim:

```
// app/controllers/contato.js

module.exports = function (app) {

  var controller = {};

  controller.listaTodos = function(req, res) {};

  controller.obtemContato = function(req, res) {};

  controller.removeContato = function(req, res) {};

  controller.salvaContato = function(req, res) {};

  return controller;
};
```

Em seguida, vamos guardar uma referência ao Model disponível na instância do Express passada como parâmetro para o módulo:

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

  var Contato = app.models.contato;

  // código omitido

};
```

Guardamos uma referência para o Model na variável Contato iniciando com letra maiúscula, uma convenção bastante utilizada para funções construtoras. Essas funções permitem utilizar o operador new para criarmos novas instâncias, objetos que representam nossos documentos.

7.7 B□□hZ□o□ o□h□□u □□ □□

Vamos iniciar pela função controller.listaTodos. Não temos interesse em instanciar um contato, apenas listar todos os cadastrados em nosso banco. Para isso, a própria função construtora Contato possui a função find. Como queremos todos os contatos, ela não receberá um critério como parâmetro.

Existem diferentes maneiras de realizarmos essa consulta, porém a função find pode retornar uma promise através da chamada encadeada à função exec:

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

  var Contato = app.models.contato;

  var controller = {}

  controller.listaTodos = function(req, res) {
    var promise = Contato.find().exec();
  };

  // código omitido
```

```
    return controller;
};
```

Aprendemos na seção 5.3 que toda *promise* possui a função then. Isso não é diferente com a *promise* do Mongoose, mas ela não possui a função catch. Vamos passar os callbacks de sucesso e de falha para a fun ção then e omitiremos a declaração da variável promise:

```
controller.listaTodos = function(req, res) {
    Contato.find().exec()
    .then(
        function(contatos) {
            res.json(contatos);
        },
        function(erro) {
            console.error(erro)
            res.status( 500).json(erro);
        }
    );
};
```

Recebemos no callback de sucesso a lista de contatos retornada e logo em seguida a enviamos através da função res.json. Já no callback de erro, logamos a informação e, antes de enviá-la para o cliente, modi□camos o status da resposta para 500 (*internal server error*).

A função find

Estamos chamando a função exec para recebermos uma *promise*. Porém, entre a chamada da função find e exec, podemos chamar outras funções do Mongoose. Por exemplo, se quisermos obter apenas o nome e o email de um contato, usamos a função select:

```
Contato.find()
.select("nome email")
.exec();
```

Se quisermos agora filtrar por parte de um e-mail, encadeamos uma chamada à função where seguida de uma das muitas funções de comparação. Em nosso caso, utilizaremos a função equals, que recebe como parâmetro uma expressão regular:

```
Contato.find()
.select("nome email")
.where("email").equals(/cont/)
.exec()
.then(function(nomeEhEmail) {
  console.log(nomeEhEmail);
});
```

Apesar de termos apenas a listagem implementada em nosso controller, já podemos testar e verificar se nossos contatos são exibidos na página principal.

7.8 Buscando pelo ID

Nosso próximo passo será implementar a busca de um contato, só que dessa vez utilizaremos a função Contato.findById:

```
controller.obtemContato = function(req, res) {
    var _id = req.params.id;
    Contato.findById(_id).exec()
    .then(
```

```
        function(contato) {
          if (!contato) throw new Error("Contato não encontrado");
          res.json(contato)
        },
        function(erro) {
          console.log(erro);
          res.status( 404).json(erro)
        }
      );
};
```

Caso o ID procurado não exista no banco, receberemos undefined como resposta. Neste caso, lançaremos uma exceção informando que o contato não foi encontrado. Como estamos usando o padrão *promise*, nosso callback de erro será invocado recebendo a exceção que lançamos. Depois de logarmos o erro, enviamos como resposta o status 404, junto do JSON com a mensagem de erro. Se tudo correr bem, enviamos o contato como resposta através de res.json.

Podemos reiniciar nossa aplicação e verificar se nossa seleção de contatos realmente funciona.

7.9 Removendo contatos

Agora atacaremos a remoção de contatos. Utilizaremos a função Contato.remove que recebe como critério o ObjectId procurado. Se a operação for realizada com sucesso, enviaremos o status padrão 200 como resposta.

```
controller.removeContato = function(req, res) {
  var _id = req.params.id;
  Contato.remove({"_id" : _id}).exec()
  .then(
    function() {
      res.end();
    },
    function(erro) {
      return console.error(erro);
```

```
    }
  );
};
```

[illegible]

Um Model criado pelo Mongoose possui a função findByIdAndRemove que remove e passa para o callback o contato removido. Em nosso caso, não temos interesse no documento removido; é por isso que a função remove foi utilizada.

Experimente reiniciar o servidor e remover alguns contatos. Tudo deve funcionar conforme o esperado.

7.10 [illegible]

Só falta implementarmos nossa lógica de gravação. Caso o ID do contato tenha sido definido, utilizaremos a função Contato.findByIdAndUpdate para atualizá-lo. Seu primeiro parâmetro é o ID do contato procurado; o segundo, os dados do contato:

```
controller.salvaContato = function(req, res) {
  var _id = req.body._id;
  if(_id) {
    Contato.findByIdAndUpdate(_id, req.body).exec()
    .then(
      function(contato) {
        res.json(contato);
      },
      function(erro) {
        console.error(erro)
        res.status(500).json(erro);
      }
    );
  } else {
    // falta ainda
```

```
  }
};
```

oZ u

Um Model do Mongoose possui a função update, porém ela apenas atualiza o documento sem retorná-lo após a modificação. A função findByIdAndUpdate, além de atualizar, nos dá acesso ao documento atualizado.

Precisamos implementar o else que adicionará um novo contato. Nesta situação, utilizaremos a função Contato.create, que recebe como parâmetro os dados do contato enviados na requisição. Diferente das funções de pesquisa, não precisamos chamar a função exec, pois a própria create já retorna uma *promise* por padrão:

```
controller.salvaContato = function(req, res) {
  var _id = req.body._id;
  if(_id) {
    Contato.findByIdAndUpdate(_id, req.body).exec()
    .then(
      function(contato) {
        res.json(contato);
      },
      function(erro) {
        console.error(erro)
        res.status( 500).json(erro);
      }
    );
  } else {
      Contato.create(req.body)
      .then(
        function(contato) {
          res.status( 201).json(contato);
        },
        function(erro) {
          console.log(erro);
```

```
            res.status(500).json(erro);
        }
      );
  }
};
```

Agora podemos reiniciar o servidor incluindo e alterando contatos e verificar se tudo funciona conforme o esperado.

7.11 Funções de persistência no Model ou no documento?

O Mongoose permite executar operações de persistência de duas maneiras. Utilizamos a primeira na seção anterior quando chamamos as funções de persistência diretamente no Model Contato. A segunda chama funções a partir de uma instância do nosso Model, que também é uma função construtora. Vejamos um exemplo desta segunda forma correspondente à inclusão de um contato:

```
var contato = new Contato(req.body);
contato.save(function(erro, contato) {
  if(erro) {
    res.status(500).end();
    console.log(erro)
  } else {
    res.json(contato);
  }
});
```

Repare que instanciamos um novo documento através do operador new, passando para nosso Model os dados do contato acessíveis através de req.body.

O problema é que funções de persistência da **instância** de um Model não retornam *promises*. Repare que no exemplo anterior somos obrigados a receber na mesma função de callback uma referência para erro e outra para o contato salvo.

7.12 C□□Z□o□□u□u□x□h□Z□

Nossa parte de persistência está pronta, porém ainda falta mais um requisito do nosso sistema. Você deve se lembrar que na seção 6.19 associamos uma emergência para um contato através do mongo shell. Chegou a hora de implementarmos a mesma associação através do Mongoose!

O primeiro passo é abrirmos o arquivo app/models/contato.js e adicionarmos a informação de emergência em nosso esquema. Contudo, seu tipo será um Schema.ObjectId:

```
var mongoose= require('mongoose');
module.exports = function() {
  var schema = mongoose.Schema({
    nome: {
      type: String,
      required: true
    },
    email: {
      type: String,
      required: true,
      index: {
        unique: true
      }
    },
    emergencia: {
      type: mongoose.Schema.ObjectId,
      ref: 'Contato'
    }
  });

  return mongoose.model('Contato', schema);
};
```

Repare que, além do type, adicionamos também a chave ref, que possui como valor um Model, em nosso caso, o Model Contato. Isso faz todo o sentido, já que temos um autorrelacionamento. Esse exemplo demonstra que podemos adotar como type qualquer Model criado por nós através do Mongoose.

Muito bem, alteramos nosso esquema, mas onde a nova informação será exibida? É o que veremos na seção a seguir.

7.13 M□o□□hZ□o□ □□□Z □u □ □Z□Z u□f□□ u□u□□x□hZ□

Precisamos adicionar na view parcial public/partials/contato.html uma caixa de seleção, permitindo que escolhamos de uma lista qual dos contatos cadastrados será a emergência. A questão é que a lista deve ser populada com os contatos cadastrados em nosso sistema.

Primeiro, vamos disponibilizar no escopo do controller public/js/controllers/ContatoController.js a lista que será acessada pela view:

```
// public/js/controllers/ContatoController.js

angular.module('contatooh').controller('ContatoController',
  function($scope, $routeParams, Contato) {

    // código anterior omitido

    Contato.query(function(contatos) {
      $scope.contatos = contatos;
    });
});
```

Agora, adicionaremos a caixa de seleção na view imediatamente após o from-group com o e-mail do contato:

```
<! -- public/partials/contato.html -->

<div class="form-group">
  <label for="emergencia">Emergência </label>
  <select id="emergencia"
          class="form-control" >
          ng-model="contato.emergencia"
  </select>
</div>
```

Repare que adicionamos na tag <select> a diretiva ng-model. Quando um contato de emergência for selecionado da lista e o usuário clicar no botão salvar, seu ObjectId será salvo diretamente em contato.emergencia. A questão agora é saber como construiremos a lista.

7.14 A o□u □ □□ □□-□□□ □□□

Vimos na seção anterior o uso da tag <select> para exibir uma *combobox* com todos os contatos cadastrados, mas ainda falta popularmos a lista. Sabemos que cada contato deverá ter sua respectiva tag <option>. Porém, no lugar de adicionarmos várias tags <option>, utilizaremos a diretiva ng-options do AngularJS responsável por popular a tag <select>:

```
<! -- public/partials/contato.html -->

<div class="form-group">
  <label for="emergencia">Emergência</label>
  <select id="emergencia"
      class="form-control" >
      ng-model="contato.emergencia"
      ng-options=
        "contato._id as contato.nome for contato in contatos"
  </select>
</div>
```

Adicionamos a diretiva ng-options, que possui comportamento parecido com ng-repeat, porém a sintaxe "contato._id as contato.nome" indica que o valor do elemento será o ObjectId do contato e o que será exibido para seleção será seu nome. O restante "for contato in contatos" percorrerá a lista de contatos disponibilizada no escopo do controller, construindo cada item de nossa lista.

Podemos salvar todas as alterações e experimentar associar algumas emergências para alguns contatos:

Fig. 7.1: Associando emergência ao contato

Porém, teremos um problema ao gravarmos um novo contato sem emergência e depois tentarmos atualizar seu nome, por exemplo. Receberemos no console a mensagem:

[TypeError: Cannot read property '_id' of undefined]

Para solucionar este problema, verificamos se o valor da chave emergência é undefined; se for, substituímos seu valor por null. Outra solução é não incluir a chave emergencia na atualização caso seu valor seja undefined:

```
controller.salvaContato = function(req, res) {
  var _id = req.body._id;

  // testando por undefined
  req.body.emergencia = req.body.emergencia || null;

  if(_id) {
    Contato.findByIdAndUpdate(_id, req.body).exec()
    .then(
      function(contato) {
        res.json(contato);
      },
      function(erro) {
        console.error(erro)
        res.status(500).json(erro);
      }
    );
  } else {
    // código comentado
```

```
    }
};
```

Somos capazes de associar emergências aos contatos, mas seria interessante exibirmos a emergência do contato em uma coluna lá na listagem de contatos. Vamos adicionar uma nova coluna em public/partials/contatos.html imediatamente após a <td> que exibe o e-mail do contato:

```
<! -- imediatamente após o th com do e-mail -->
<th>Emergência</th>
...
<! -- imediatamente após a td com o e-mail do contato -->
 <td>{{contato.emergencia.nome}}</td>
```

Terminamos! Porém, ao recarregarmos a página, nenhum dos contatos com emergência associada exibe a informação em sua coluna. Por quê? É o que veremos na próxima seção.

7.15 A [illegible]

Quando de[illegible]nimos uma referência em nosso esquema, apenas damos uma pista para o Mongoose de como ele deve buscá-la, porém temos que explicitar isso em nosso código através da função populate.

Em nosso controller app/controllers/contato.js, temos a fun ção listaTodos. Nela, utilizamos a fun ção Contato.find para listar todos os contatos cadastrados e que, no [illegible]nal, retorna uma *promise* através da chamada à função exec. Precisamos chamar a populate antes de retornamos a *promise*, isto é, antes da chamada à exec:

```
// app/controllers/contato.js

// anterior omitido
Contato.find().populate('emergencia').exec()
// código posterior omitido
```

Repare que passamos para a função populate a referência que o Mongoose deverá resolver para nós. No [illegible]nal, cada contato de nossa lista, no lugar

de conter apenas o ObjectId da sua emergência, conterá a emergência inteira como um embedded document. Agora que temos um documento completo, podemos acessar qualquer uma de suas propriedades em nossa view:

Fig. 7.2: Exibindo contato de emergência

Perfeito! Mais uma funcionalidade implementada com o auxílio do Mongoose.

Acesso liberado para todos?

Aprendemos a realizar operações de consulta, inclusão, remoção e atualização através do Mongoose, e assim terminamos a parte de persistência da nossa aplicação. Porém, nosso sistema está aberto para qualquer um que saiba sua URL. Precisamos garantir que apenas usuários autenticados possam utilizar nossa aplicação e é exatamente este assunto que abordaremos no próximo capítulo.

CZ£ –¶ 8

Autenticação com Passport

"Aprender sem pensar é esforço vão; pensar sem nada aprender é nocivo"
– Confúcio

Precisamos garantir que apenas usuários autenticados tenham acesso à nossa aplicação, porém não queremos armazenar dados sensíveis como senhas por uma questão de segurança. Também não desejamos implementar nosso próprio servidor de autenticação.

Uma solução é utilizarmos serviços de autenticação gratuitos fornecidos na web como os do Facebook, do Twitter entre outros. Utilizaremos o serviço do **GitHub** (https://github.com/) .

C□□□ Z □□ G□□ H□f

Caso você não tenha uma conta no Github, esta é a hora de criá-la em https://github.com. O registro é gratuito para repositórios públicos.

O GitHub utiliza o protocolo OAuth 2.0 para autorização. Antes de entrarmos nos detalhes de sua implementação em nossa aplicação, faremos um apanhado geral de como o protocolo funciona.

8.1 OA□□ □ 2.0

OAuth 2.0 é um protocolo aberto para autenticação (quem você é) e autorização (o que você pode fazer) que permite aplicações acessarem os dados uma das outras. Ele veio substituir a versão 1.0 criticada pela sua complexidade e uso de certi□cados. OAuth 2.0 utiliza apenas SSL/TSL.

Papéis envolvidos

Há quatro papéis envolvidos durante o □uxo do OAuth 2.0. São eles:

- **DONO DO RECURSO**: usuário ou aplicação cujos recursos deseja compartilhar.
- **SERVIDOR DE RECURSOS**: local dos recursos do DONO DO RECURSO.
- **APLICAÇÃO CLIENTE**: acessa recursos do DONO DO RECURSO em um SERVIDOR DE RECURSOS.
- **SERVIDOR DE AUTORIZAÇÃO**: autoriza a APLICA˙ˆO CLIENTE a ter acesso aos recursos do DONO DO RECURSO em um SERVIDOR DE RECURSOS.

O fluxo do protocolo

O OAuth 2.0 segue um fluxo bem definido. Vejamos isso no contexto da nossa aplicação:

1) O usuário não autenticado tenta acessar a APLICAÇÃO CLIENTE Contatooh.

2) A APLICAÇÃO CLIENTE redireciona o usuário para a página de login do GitHub, porém enviando na requisição o **CLIENT_ID** da aplicação. Dessa maneira, o SERVIDOR DE AUTORIZAÇÃO sabe quem está tentando acessar o recurso protegido.

3) O usuário se loga no GitHub e, logo em seguida, é enviado através de uma **REDIRECT_URI** novamente para a APLICAÇÃO CLIENTE, inclusive um CÓDIGO DE AUTENTICAÇÃO é enviado na requisição.

4) Por baixo dos panos, a APLICAÇÃO CLIENTE se conecta diretamente ao SERVIDOR DE AUTORIZAÇÃO enviando seu **CLIENT_ID**, **CLIENT_PASSWORD** e o CÓDIGO DE AUTENTICAÇÃO recebido no redirecionamento.

5) O SERVIDOR DE AUTENTICAÇÃO devolve para a APLICAÇÃO CLIENTE um CONSENTIMENTO DE AUTORIZAÇÃO que permite acesso às informações do DONO DO RECURSO.

Agora que já temos uma ideia de como o protocolo OAuth 2.0 funciona, aprenderemos na próxima seção a registrar nossa aplicação Contatooh no provedor de autenticação escolhido.

8.2 Ru [illegible] o ou Z u hZl e

Utilizaremos o GitHub para autenticar nossos usuários que devem possuir uma conta neste serviço. Precisamos registrar nossa aplicação Contatooh no próprio site (https://github.com/) . Geralmente, criamos um usuário específico para a aplicação e nele registramos a aplicação, mas nada impede que você use a sua mesma conta para essa finalidade.

Acessando profile

Depois de logar em sua conta no GitHub, no canto superior direito há uma engrenagem que, ao ser clicada, o levará diretamente para seu pro□le:

Fig. 8.1: Github: atalho pro□le

Registrando nova aplicação

Na página de pro□le, no canto esquerdo, há um painel com uma série de links. O link **Applications** exibe uma tela com as aplicações já registradas e o botão **Register new Application**, para que possamos cadastrar nossa aplicação. Na tela de registro, você precisará informar:

- **Application name**: podemos utilizar qualquer nome, em nosso caso será **contatooh**.
- **Homepage URL**: como nossa aplicação web ainda roda localmente, preencha com **http://localhost:3000**.
- **Application Description**: coloque uma breve descrição da sua app ou deixe em branco.
- **Authorization callback URL**: o endereço da nossa aplicação que será chamado pelo GitHub após a autenticação. Vamos utilizar o endereço **http://localhost:3000/auth/github/callback**.

Por □m, clique no botão **Register application** para □nalizar o registro. Com a aplicação registrada, no canto superior direito será exibido o **CLIENT ID** e o **CLIENT SECRET**. Tome nota dessas informações pois elas serão utilizadas mais à frente:

Sua instalação é feita através do npm. Só não esqueça de executá-lo dentro da pasta raiz do projeto, local do nosso arquivo package.json:

```
npm install passport@0.2 --save
```

A inicialização do Passport depende de o Express estar configurado para trabalhar com sessões, algo que ainda não fizemos. É por isso que instalaremos os módulos express-session e o cookie-parser:

```
npm install express-session@1.7 cookie-parser@1.3 --save
```

Precisamos alterar agora nosso arquivo config/express.js para ativarmos os *middlewares* de cookie, sessão e de inicialização do Passport:

```
// config/express.js

var cookieParser = require('cookie-parser');
var session = require('express-session');
var passport = require('passport');

// código abaixo vem antes do carregamento de rotas

app.use(cookieParser());
app.use(session(
    { secret: 'homem avestruz',
      resave: true,
      saveUninitialized: true
    }
));
app.use(passport.initialize());
app.use(passport.session());

// código posterior omitido
```

O primeiro middleware cookieParser realiza o parser do header de cookies da requisição populando req.cookies e armazena o ID da sessão. O segundo middleware session cria por padrão a sessão do usuário em memória. Ele recebe três parâmetros:

- **secret**: o cookie de sessão é assinado com este segredo para evitar adulteração.
- **resave**: garante que as informações da sessão serão acessíveis através de cookies a cada requisição.
- **saveUnitialized**: essa opção soluciona problemas que envolvem a requisição de uma permissão antes de atribuir um cookie.

Quando usamos Express, precisamos chamar a função passport.initialize para inicializar o Passport. Como nossa aplicação usa sessões de login persistentes, também precisamos utilizar o middleware passport.session.

Um ponto a destacar é que a inicialização da sessão do Express deve vir sempre antes de passport.session para garantirmos que a sessão de login seja restaurada na ordem correta.

> **[illegible]**
>
> No Express, o desenvolvedor precisa saber exatamente em qual posição da pilha configurar um novo middleware e seu impacto nos demais. Quanto maior o número de middlewares, mais difícil será sua tarefa. Essa complexidade é apelidada de **middleware hell**.

Antes que possamos utilizar o Passport para autenticar nossas requisições, uma estratégia de autenticação precisa ser configurada.

8.4 [illegible]

O Passport possui várias estratégias de autenticação, inclusive podemos criar novas estratégias. Porém, já definimos que nos autenticaremos através do GitHub e, por isso, baixaremos uma estratégia específica para este provedor de autorização:

```
npm install passport-github@0.1 --save
```

Precisamos passar algumas configurações para nossa estratégia e registrá-la no Passport. Criaremos um módulo com esta finalidade. Primeiro criaremos o arquivo config/passport.js e nele importaremos o Passport e a estratégia de autenticação:

```
// config/passport.js
var passport = require('passport');
var GitHubStrategy = require('passport-github').Strategy;
```

Utilizaremos como estratégia de autenticação uma instância de GitHubStrategy que recebe dois parâmetros. O primeiro é um objeto com as chaves clientID, clientSecret, callbackURL com seus respectivos valores disponibilizados na configuração de nossa aplicação no GitHub. O segundo parâmetro é um callback que será **chamado apenas uma vez quando o usuário se autenticar**:

```
// config/passport.js

// código anterior omitido

module.exports = function() {

    // código anterior omitido

    passport.use(new GitHubStrategy({
        clientID: 'SEU CLIENT ID',
        clientSecret: 'SEU CLIENT PASSWORD',
        callbackURL: 'SUA REDIRECT_URI'
    }, function(accessToken, refreshToken, profile, done) {

    }));
};
```

Repare que o callback recebe quatro parâmetros, mas o que realmente nos interessa são os dois últimos: profile e done. O primeiro é um objeto que representa o perfil do usuário no GitHub. O segundo, uma função de callback que deve receber como parâmetro os dados que queremos mais tarde armazenar na sessão. Que dados serão esses? É o que veremos na próxima seção.

8.5 Du □□□□o Sh□u □Z o U□□□□□

Em uma aplicação típica, associamos o pro□le do GitHub com um usuário em nosso banco, retornando-o no lugar do pro□le para a função done da estratégia que vimos na seção anterior. É isso que faremos.

Criaremos primeiro o arquivo app/models/Usuario.js, no qual de□niremos um esquema para, logo em seguida, disponibilizarmos um Model a partir dele:

```
// app/models/Usuario.js

var mongoose= require('mongoose');
module.exports = function() {
  var schema = mongoose.Schema({
    login: {
      type: String,
      required: true,
      index: {
        unique: true
      }
    },
    nome: {
      type: String,
      required: true,
    },
    inclusao: {
      type: Date,
      default: Date.now
    }
  });
  return mongoose.model('Usuario', schema);
};
```

Repare que temos uma novidade. Utilizamos o tipo Date para a chave inclusao e adotamos como valor padrão a data atual do sistema. As novidades ainda não acabaram, veremos mais uma com Mongoose na próxima seção.

8.6 M□□□□□u u □□□□□□

Criamos na seção anterior o Model Usuario para que possamos realizar operações de consulta e persistência em nosso banco. Sabemos que a função de callback passada para nossa estratégia de autenticação será chamada apenas uma vez quando o usuário se autenticar e que nela temos acesso ao seu per□l no GitHub através do parâmetro profile. É neste momento que buscaremos o usuário em nosso banco.

Utilizaremos como critério de busca profile.username, criando um novo usuário no banco caso ele não exista. No □nal, passaremos o usuário existente ou recém-criado como parâmetro para a função done concluindo o processo.

Sabemos implementar essa tarefa utilizando as funções find e create, porém queremos algo mais direto como findOrCreate, justamente uma função que o Mongoose não implementa por padrão. Podemos resolver isso facilmente utilizando o sistema de plugins do Mongoose. Primeiro, através do npm instalaremos o plugin:

```
npm install mongoose-findorcreate@0.1 --save
```

Agora, basta importamos o plugin associando-o com Schema:

```
// app/models/Usuario.js

var mongoose = require('mongoose');

// importando o plugin
var findOrCreate = require('mongoose-findorcreate')

    // código da declaração do esquema omitido

    // associando plugin ao nosso esquema
    schema.plugin(findOrCreate);
    return mongoose.model('Usuario', schema);
};
```

Excelente! Agora, já podemos implementar nossa lógica que busca ou cria um usuário para nós em nosso banco, disponibilizando no □nal o usuário encontrado ou criado. Mas temos um problema: nosso módulo de con□guração

do Passport não possui uma referência para o Express. Como teremos acesso ao Model Usuario? Simples, podemos solicitar ao próprio Mongoose um Model através da função mongoose.model passando o nome do Model como parâmetro. Para isso, precisaremos importar o Mongoose em nosso módulo:

```
// config/passport.js

// código anterior omitido

var mongoose = require('mongoose');

module.exports = function() {

    var Usuario = mongoose.model('Usuario');

    // código anterior omitido

    passport.use(new GitHubStrategy({
        clientID: 'SEU CLIENT ID',
        clientSecret: 'SEU CLIENT PASSWORD',
        callbackURL: 'SUA REDIRECT_URI'
    }, function(accessToken, refreshToken, profile, done) {

        Usuario.findOrCreate(
            { "login" : profile.username},
            { "nome" : profile.username},
            function(erro, usuario) {
                if(erro)
                    console.log(erro);
                    return done(erro);
                }
                return done(null, usuario);
            }
        );
    }));
};
```

P□□□□□u □

O plugin findOrCreate não tem suporte a *promises*.

A função Usuario.findOrcreate recebe como primeiro parâmetro profile.username do GitHub, nosso critério de busca. O usuário será retornado caso o critério seja atendido, caso contrário, será adicionado um novo em nosso banco, porém o padrão é incluir apenas a chave do critério de busca como informação. É por isso que adicionamos o segundo parâmetro, que contém os dados complementares, em nosso caso, apenas a chave nome, que receberá também como valor o profile.username do GitHub. No □nal, passamos o usuário que veio do nosso banco para a função done.

P□□□□□u u □□□□Z□□□l e□

O Passport padroniza o objeto profile disponibilizando chaves comuns como username e displayName independente da estratégia utilizada.

O que faremos com essa informação disponibilizada para o Passport? Gravaremos diretamente na sessão? E como faremos para acessá-la em nossas requisições? Precisamos dar uma pista para o Passport de como lidar com essas situações.

8.7 Su □□Z□□□l e□ u ou □□u □□Z□□□l e□ o□ □□□□□□

Precisamos indicar para o Passport como a informação do usuário será **serializada** na sessão, processo realizado apenas uma vez quando o usuário se autenticar. Não podemos simplesmente persistir o usuário totalmente na sessão. Imagine se novas informações fossem adicionadas?

Serializando o ObjectID do usuário na sessão

Não queremos correr o risco de onerar a memória do servidor guardando informações desnecessárias, motivo pelo qual serializaremos apenas o ObjectId do usuário na sessão. Esse processo é realizado através da função passport.serializeUser, que recebe dois parâmetros. O primeiro é o usuário que foi passado pela estratégia de autenticação. O segundo, uma função que recebe a informação do usuário que desejamos serializar na sessão:

```
// config/passport.js
// código anterior omitido

module.exports = function() {

    // código anterior omitido

    /*
      Chamado apenas UMA vez e recebe o usuário do nosso
      banco disponibilizado pelo callback da estratégia de
      autenticação. Realizará a serialização apenas do
      ObjectId do usuário na sessão.
    */

    passport.serializeUser(function(usuario, done) {
      done(null, usuario._id);
    });
};
```

Porém, toda vez que precisarmos acessar outras informações do usuário em nossos controllers, será necessário buscá-lo no banco através de seu ObjectId. A boa notícia é que podemos delegar esse processo de **desserialização** ao Passport.

Desserializando o usuário da sessão

Em cada requisição, o Passport chamará sua função de desserialização passando como parâmetro o ObjectId do usuário armazenado na sessão.

Usaremos essa informação para buscar o usuário no banco, inclusive o Passport disponibilizará o usuário encontrado em cada request através da chave req.user, tornando-o facilmente acessível em nossos controllers:

```
// config/passport.js
// código anterior omitido

module.exports = function() {

    // código anterior omitido

    // Recebe o ObjectId do usuário armazenado na sessão
    // Chamado a CADA requisição

    passport.deserializeUser(function(id, done) {
      Usuario.findById(id).exec()
      .then(function(usuario) {
          done(null, usuario);
      });
    });
};
```

Até agora, inicializamos o Passport com o Express, configuramos sua estratégia de autenticação e implementamos as funções de serialização e desserialização do usuário. Porém, ainda não estamos protegendo os recursos da nossa aplicação, assunto que veremos a seguir.

8.8 Protegendo rotas

Ainda falta definirmos as rotas envolvidas no processo de autenticação, principalmente aquela que protegerá nossos recursos. Vamos criar o arquivo app/routes/auth.js:

```
// app/routes/auth.js

module.exports = function(app) {

    app.get('/auth/github', passport.authenticate('github'));
```

```
    app.get('/auth/github/callback',
            passport.authenticate('github', {
        successRedirect: '/'
    }));
}
```

A primeira rota /auth/github redirecionará o usuário para a página de login do GitHub enviando por baixo dos panos o CLIENT ID da aplicação. Repare que utilizamos como controller o retorno da função passport.authenticate('github'), que saberá lidar com a requisição.

A segunda rota /auth/github/callback possui o mesmo identificador que cadastramos como Authorization callback URL no GitHub. Quando o usuário se logar, o GitHub chamará a rota passando o CÓDIGO DE AUTORIZAÇÃO. Com o código recebido, o Passport se comunicará por baixo dos panos com o SERVIDOR DE AUTENTICAÇÃO que protege o profile do usuário no GitHub, solicitando o CONSENTIMENTO DE AUTORIZAÇÃO para daí acessar o profile. Se a autenticação for bem-sucedida, o usuário será direcionando para nossa aplicação Contatooh.

Verificando autenticação do usuário

Nossa solução ainda não está completa, precisamos direcionar para a página de login qualquer usuário não autenticado.

O Passport convenientemente disponibiliza na requisição a função isAutenticated, que nos permite saber em nossos controllers se o usuário está autenticado ou não. Se ele estiver autenticado, deixaremos que as próximas rotas definidas sejam processadas, caso contrário, ele será redirecionado para uma página com apenas o link "Entre pelo GitHub". Essa página será uma view ejs, logo, só poderá ser acessada através de uma rota específica.

Mas qual rota utilizaremos? Criaremos uma que será sempre ativada independente do identificador do recurso utilizado. Conseguimos isso facilmente com o identificador /:

```
// app/routes/auth.js
// código anterior omitido
```

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

    // código anterior omitido

    app.get('/', function(req, res, next) {
        if(req.isAuthenticated()) {

            // permite que outras rotas sejam processadas
            return next();
        } else {

            // renderiza auth.ejs
            res.render( "auth");
        }
    });
};
```

Repare que na própria declaração da rota já implementamos seu controller através de uma função anônima, não havendo a necessidade de criar um novo arquivo .js. Criaremos agora nossa view app/views/auth.ejs:

```
<!-- app/views/auth.ejs -->

<!docytype html>
<html>
  <head>
    <title>Autenticação do Usuário</title>
    <meta name="viewport" content="width=device-width">
    <meta charset= "UTF-8">
    <link rel="stylesheet"
        href="vendor/bootstrap/dist/css/bootstrap.css">
    <link rel="stylesheet"
        href="vendor/bootstrap/dist/css/bootstrap-theme.css">
  </head>
  <body>
      <div class="container">
          <a href="/auth/github">
```

```
                <h1 class="text-center">Entre pelo GitHub</h1>
            </a>
        </div>
    </body>
</html>
```

Repare que o link “Entre pelo GitHub” aponta para a rota auth/github, aquela de nossa aplicação responsável pelo redirecionamento e envio do CLIENT ID da aplicação para o SERVIDOR DE AUTORIZAÇÃO do GitHub.

O problema é que precisaenms2ee(r)-1002(m)4ea que nossa rota vs-5.995i(Ôcv)-3.00

```
                app.get('port'));
});
```

Antes de testarmos o resultado, tenha certeza de que não esteja logado no GitHub para que possamos ver todas as etapas acontecendo.

Depois de reiniciar o servidor, tente acessar a aplicação. Você deverá ser redirecionado para a view `auth.ejs` com o link "Entre pelo GitHub", que ativará a rota do passport responsável pelo envio do `CLIENT ID` de sua aplicação para o serviço de autenticação do GitHub.

Fig. 8.3: Solicitação de login

Depois de clicar no link e realizar o processo pela primeira vez, o SERVIDOR DE AUTORIZAÇÃO do GitHub solicitará uma confirmação para que a aplicação Contatooh acesse o recurso. Não se preocupe, essa confirmação será realizada apenas uma vez.

Fig. 8.4: Autorização de acesso da aplicação à sua conta no GitHub

Implementamos autenticação em nosso sistema através de OAuth 2.0 com ajuda do Passport, mas ainda falta exibirmos uma informação muito comum em sistemas: o usuário logado.

8.9 C□□f□□Z□o□□u□o□□u□□b□□/h□□u□□u

Precisamos exibir o usuário logado em nossa aplicação, inclusive podemos até criar uma rota com esta informação acessível por nosso cliente em AngularJS. Porém, aprendemos na seção 2.6 a trabalhar com templates ejs processados no lado do servidor. Esta é uma boa hora de fazermos as pazes com esta tecnologia e aproveitarmos o melhor dos dois mundos. A página index.ejs continuará sendo uma view do AngularJS, porém o navegador já a receberá com a informação do usuário logado.

Nosso primeiro passo será mover nossa página public/index.html para a pasta app/views, renomeando-a para index.ejs. Em seguida, adicionaremos uma lacuna em nosso template indicando que ele precisa da informação usuarioLogado:

```
<!-- app/views/index.ejs -->

<! -- antes da div com a diretiva ng-view -->
<div class="text-right">
    <p>
        Logado como: <span id="usuario-logado">
                        <%=usuarioLogado %>
                    <span>
    </p>
</div>
```

Como não temos mais a página public/index.html, a URL http://localhost:3000 deixará de funcionar. Vamos criar a rota / para recuperarmos este padrão, devolvendo nossa view app/views/index.ejs com a informação do usuário logado já preenchida. Vamos criar o arquivo app/routes/index.ejs, que fará todo esse trabalho para nós. Não precisamos nos preocupar em associá-lo com o Express porque estamos utilizando o express-load, que já o carregará automaticamente:

```
// app/routes/index.js

module.exports = function(app) {
    app.get('/', function(req, res) {
        res.render( 'index', { "usuarioLogado" : req.user.login});
```

```
    });
};
```

Repare que estamos buscando na requisição a informação do usuário logado. Lembre-se que essa informação sempre estará disponível por causa do processo de desserialização do usuário da sessão realizado pelo Passport.

Já podemos testar reiniciando nosso servidor e acessando a URL http://localhost:3000. Tudo deve continuar funcionando, a única diferença é a exibição do usuário logado no topo direito de nossa página.

Fig. 8.5: Usuário logado

8.10 Implementando Logout

Nossa autenticação está funcionando, inclusive exibimos o usuário logado em nossa página, porém falta ainda implementarmos a funcionalidade de logout.

Vamos editar nossa rota app/routes/auth.js adicionando a rota /logout. Nela, chamaremos a função req.logOut adicionada automaticamente pelo Passport a cada requisição. Por fim, redirecionaremos o usuário para a página principal através da rota /:

```
// app/routes/auth.js
// código anterior omitido

module.exports = function(app) {

    // código anterior omitido
```

e experimente acessar diretamente pelo navegador nosso *REST Endpoint* que devolve uma lista de contatos:

http://localhost:3000/contatos

Para nosso espanto, ele continua acessível, mesmo para usuários não autenticados. Isso acontece porque não estamos verificando em nossa API REST a autenticação do usuário, mas imagine consultar `req.isAutenticated` em cada um de nossos controllers? Com certeza estaríamos colocando uma responsabilidade que não lhe dizem respeito.

A boa notícia é que podemos definir mais de uma função de callback para o tratamento de rotas. Em nosso arquivo app/controllers/contato.js, criaremos uma função com a responsabilidade de verificar se o usuário está autenticado. Se ele estiver, damos a chance para que o próximo callback processe a requisição, caso contrário, enviaremos como resposta "Não autorizado" com o status 401 (*Unauthorized*):

```
// app/controllers/contato.js

function verificaAutenticacao(req, res, next) {
  if (req.isAuthenticated()) {
    return next();
  } else {
    res.status('401').json('Não autorizado');
  }
}
```

Em seguida, passaremos a função como primeiro callback para o tratamento das rotas:

```
// app/controllers/contato.js
// código anterior omitido

module.exports = function (app) {
    var controller = app.controllers.contato;

    app.route('/contatos')
```

```
        .get(verificaAutenticacao, controller.listaTodos)
        .post(verificaAutenticacao, controller.salvaContato);

    app.route('/contatos/:id')
        .get(verificaAutenticacao, controller.obtemContato)
        .delete(verificaAutenticacao, controller.removeContato);
};
```

Você já deve ter percebido que rotas do Express também são middlewares e que a ordem faz toda a diferença. Em nosso caso, além da importância da ordem de declaração das rotas, elas recebem agora dois parâmetros. A novidade é que o segundo callback passado como parâmetro, nosso controller, será chamado apenas se o usuário estiver autenticado.

Podemos deixar ainda melhor nosso código isolando o código de verificação da autenticação em seu próprio módulo. Faz todo sentido, porque ela poderá ser reaproveitada por outros controllers de nossa aplicação, não apenas app/controllers/contato.js. Vamos criar o módulo contatooh/config/auth.js, que devolve nossa função de verificação no lugar de um objeto:

```
// contatooh/config/auth.js

module.exports = function(req, res, next) {
  if (req.isAuthenticated()) {
    return next();
  } else {
    res.status('401').json('Não autorizado');
  }
};
```

Agora, vamos alterar app/controllers/contato.js, que no final deverá estar assim:

```
var verificaAutenticacao = require('../../config/auth');

module.exports = function (app) {
    var controller = app.controllers.contato;
```

```
    app.route('/contatos')
        .get(verificaAutenticacao, controller.listaTodos)
        .post(verificaAutenticacao, controller.salvaContato);

    app.route('/contatos/:id')
        .get(verificaAutenticacao, controller.obtemContato)
        .delete(verificaAutenticacao, controller.removeContato);
};
```

Pronto! Experimente acessar (http://localhost:3000/contatos) sem estar autenticado. Você deve receber a mensagem "Não autorizado".

Agora podemos ir além da autenticação e tornar nossa aplicação ainda mais segura, assunto do próximo capítulo.

CZ£ –¶ 9

Tornando sua aplicação ainda mais segura

"Age sempre de tal modo que o teu comportamento possa vir a ser princípio de uma lei universal."
– Emmanuel Kant

Este capítulo demonstrará como é possível aplicar pequenas modi□cações em nossa aplicação tornando-a ainda mais segura. São elas:

No Express:

- Adicionar ou remover informações do *header* das requisições;
- Evitar "vazar" a tecnologia utilizada.

No MongoDB:

- Prevenir *query selector injection*;
- Substituir a estratégia *document replace*.

Vamos começar controlando as informações do header das requisições.

9.1 Hu u: oou ou ou Z l Z

Um ponto importante para garantir a segurança de nossa aplicação é controlando as informações presentes no header das requisições. Veremos que a omissão de um header ou a adição de outro pode trazer benefícios para a segurança.

Nada nos impede de pesquisarmos na internet quais informações removeremos ou adicionaremos em nosso header, porém foi criado um projeto com esta nalidade: o **Helmet**.

O Helmet (https://github.com/evilpacket/helmet) possui uma coletânea de middlewares de tratamento de header já prontos para uso, tornando nossa aplicação mais segura. Sua instalação é feita como qualquer outro módulo do Node.js:

```
npm install helmet@0.4 --save
```

Se quisermos habilitar todos os seus middlewares, basta adicionarmos:

```
// config/express.js
helmet = require('helmet');

// código anterior omitido

// imediatamente após a inicialização da sessão do Passport
app.use(helmet());

// código posterior omitido
```

Mas quais são os middlewares ativados? Será que realmente precisamos ativar todos eles? Eles protegem contra qual ataque? Existe alguma limitação?

O ideal é ativarmos apenas aqueles middlewares que façam sentido dentro do contexto da nossa aplicação, entendendo qual tipo de ataque eles protegem. Vejamos os problemas que podem acometer a MEAN Stack e quais middlewares do Helmet utilizar.

Esconder X-Powered-By

Não é incomum aplicações adicionarem no header http a informação ** X-Powered-By ** indicando qual tecnologia está sendo utilizada pelo servidor. Hackers podem utilizar essa informação para tentar explorar vulnerabilidades conhecidas da tecnologia utilizada. Podemos resolver isso facilmente no Express deixando de informar o header X-Powered-By:

```
// imediatamente após a inicialização da sessão do Passport
app.disable('x-powered-by');
```

Ou você pode querer dificultar ainda mais a vida do potencial invasor:

```
// imediatamente após a inicialização da sessão do Passport
app.use(helmet.hidePoweredBy({ setTo: 'PHP 5.5.14' }));
```

Nesse código, fornecemos uma informação falsa através do middleware helmet.hidePoweredBy:

Aplicação acessível indevidamente dentro de um <frame> ou <iframe>

Nem sempre você quer que sua aplicação seja colocada dentro de um <frame> ou <iframe>, evitando possíveis ataques do tipo clickjacking. Esse ataque disponibiliza um <iframe> invisível ou redimensionado contendo a página que o atacante quer que visitemos sem saber, porém ele é colocado sobre algum elemento da página como links e botões aparentemente inofensivos. Quando clicarmos sobre o link ou botão, estaremos executando um código da página do <iframe> e não da página no qual nos encontramos. Porém, podemos resolver isso:

```
// imediatamente após a inicialização da sessão do Passport
app.use(helmet.xframe());
```

É através do middleware helmet.xframe

evilpacket/helmet) para conhecer outros e avaliar se fazem sentido dentro da MEAN Stack. Por fim, teremos em nosso arquivo config/express.js os seguintes middlewares ativados:

```
helmet = require('helmet');
// código anterior omitido

// imediatamente após a inicialização da sessão do Passport
app.use(helmet.xframe());
app.use(helmet.xssFilter());
app.use(helmet.nosniff());
app.disable('x-powered-by');
```

Vimos questões que afetam o Express, mas há alguma que afeta diretamente o MongoDB? É o que veremos na próxima seção.

9.2 MongoDB/API REST: um olhar mais atento

Um ponto bastante destacado do MongoDB é o fato de ele ser *injection free*, diferente de bancos SQL nos quais o ataque *SQL injection* é possível, dependendo de como o desenvolvedor estrutura seu código.

Mas será que estamos mesmo livres de injection? Vamos revisitar nossa REST API que remove um contato do banco pelo seu ObjectId:

```
// app/controllers/contato.js
// código anterior omitido

controller.removeContato = function(req, res) {
  var _id = req.params.id;
  Contato.remove({"_id" : _id}).exec()
  .then(
    function() {
      res.end();
    },
    function(err) {
      return console.error(erro);
    }
```

```
    );
};
// código posterior omitido
```

Nossa API espera receber como parâmetro uma string contendo o ObjectId do contato, porém o que acontecerá se, no lugar de passarmos uma string, passarmos um objeto com o *query selector* $ne que vimos na seção 6.12? Por exemplo:

```
{ "$ne" : null}
```

Bem, teríamos como critério final para nossa consulta:

```
{"_id" : { "$ne" : null}}
```

O parâmetro anterior resultaria na remoção de **todos** os contatos de nosso banco! Essa mesma estratégia poderia ser utilizada em outros lugares que esperam um critério de consulta.

Para solucionarmos o problema de injeção através de *query selectors*, devemos aceitar apenas strings como critério. Porém, se recebermos um objeto, basta remover qualquer chave que contenha $ como valor. Há um projeto que realiza essa verificação para nós, o **mongo-sanitize** (https://github.com/vkarpov15/mongo-sanitize) . Sua instalação é feita através do npm:

```
npm install mongo-sanitize@1.0 --save
```

Agora só precisamos alterar app/controllers/contato.js e sanitizar req.params.id:

```
// app/controllers/contato.js

var sanitize = require('mongo-sanitize');

// código anterior omitido

controller.removeContato = function(req, res) {
  var _id = sanitize(req.params.id);
  Contato.remove({"_id" : _id}).exec()
  .then(
```

```
      function() {
        res.end();
      },
      function(erro) {
        return console.error(erro);
      }
    );
};
```

Pronto! Chaves que contenham query selectors serão removidas do objeto.

9.3 Evitando ataques de Cross Site Scripting

Aprendemos na seção 6.16 o quanto é conveniente para o desenvolvedor utilizar document replace como estratégia de atualização de documentos, inclusive ela foi colocada em prática na seção 7.10. Porém, em uma aplicação que não seja um protótipo, esta estratégia pode causar efeitos indesejados.

Vamos revisitar a parte do código de nosso controller app/controllers/contato.js que atualiza ou cria um contato:

```
// app/controllers/contato.js

// código anterior omitido

controller.salvaContato = function(req, res) {
  var _id = req.body._id;
  if(_id) {
    Contato.findByIdAndUpdate(_id, req.body).exec()
    .then(
      function(contato) {
        res.json(contato);
      },
      function(erro) {
        console.error(erro)
        res.status( 500).json(erro);
      }
    );
```

```
  } else {
    Contato.create(req.body)
    .then(
      function(contato) {
        res.status(201).json(contato);
      },
      function(erro) {
        console.log(erro);
        res.status(500).json(erro);
      }
    );
  }
};
// código posterior omitido
```

Veja que nos dois casos contamos que req.body conterá dados do contato. Porém, nada impede que alguém injete indevidamente nos dados do post outras informações que não dizem respeito ao contato. Por exemplo:

```
// req.body
{ "nome": "X", "email" : "Y", "emergencia": "Z",
  "outra-chave" : "W"};
```

Como estamos utilizando Mongoose, o ODM ignorará chaves que não façam parte do esquema, excelente! Porém, imagine se nosso contato, além das chaves que acabamos de ver tivesse outra que armazena o número de vezes que o contato foi contactado, informação que não está presente na tela de cadastro:

```
// req.body
{ "nome": "X", "email" : "Y", "emergencia": "Z",
  "contactado" : 500000};
```

Como a chave contactado faz parte do esquema, o Mongoose a atualizará gerando inconsistência de nossos dados.

Podemos resolver isso facilmente pinçando de req.body apenas as chaves que fazem sentido dentro daquele contexto:

```
// app/controllers/contato.js
```

```
// código anterior omitido

controller.salvaContato = function(req, res) {
  var _id = req.body._id;
  /*
    Independente da quantidade de parâmetros,
    apenas selecionamos o nome, email e emergencia:
  */
  var dados = {
    "nome" : req.body.nome,
    "email" : req.body.email,
    "emergencia" : req.body.emergencia || null
  };

  if(_id) {
    Contato.findByIdAndUpdate(_id, dados).exec()
    .then(
      function(contato) {
        res.json(contato);
      },
      function(erro) {
        console.error(erro)
        res.status( 500).json(erro);
      }
    );
  } else {
    Contato.create(dados)
    .then(
      function(contato) {
        res.status( 201).json(contato);
      },
      function(erro) {
        console.log(erro);
        res.status( 500).json(erro);
      }
    );
  }
};
// código posterior omitido
```

Repare que, por mais que seja interessante utilizar na íntegra os dados enviados na requisição, é altamente recomendado realizarmos um filtro antes, mesmo que isso aumente nossa impedância.

9.4 Tratando 404

Na seção 9.1, aprendemos a omitir o header X-Powered-By e a mascará-lo se assim desejássemos. A ideia era não deixar que terceiros soubessem a tecnologia utilizada pela nossa aplicação, evitando que vulnerabilidades específicas da tecnologia fossem exploradas.

Mas será que ainda estamos dando alguma pista da tecnologia utilizada? Vamos experimentar acessar uma página que não existe:

http://localhost:3000/pagina-que-nao-existe

Você receberá como resposta a seguinte mensagem:

Cannot GET /pagina-que-nao-existe

A mensagem é padrão do Express, dessa forma, informando para alguém mais atento que estamos utilizando este framework.

Podemos resolver isso facilmente criando uma rota com o identificador *, ativado para qualquer endereço, até aqueles que não tiverem uma rota equivalente. Porém, ele deve vir obrigatoriamente como última rota da aplicação, dando chance para que outras sejam processadas.

Vamos editar o arquivo config/express.js adicionando a rota imediatamente abaixo da chamada do express-load:

```
// código anterior omitido
load('models', {cwd: 'app'})
    .then('controllers')
    .then('routes/auth.js')
    .then('routes')
    .into(app);

// se nenhum rota atender, direciona para página 404
app.get('*', function(req, res) {
```

```
    res.status(404).render('404');
});

// código posterior omitido
```

Você deve se perguntar o porquê de ela não estar em um arquivo em separado. Precisamos ter a garantia de que ela seja a última rota processada e teremos essa certeza colocando-a imediatamente após o carregamento de rotas com express-load.

Repare que adicionamos o status 404 e depois renderizamos a view 404. Utilizamos uma view ejs para evitar que a página seja chamada publicamente, ou seja, ela deve ser chamada apenas através de nossa rota caso a URL acessada não seja encontrada em nosso servidor. Precisamos criá-la em app/views/404.ejs:

```
<!doctype html>
<html>
  <head>
    <title>Página não encontrada</title>
    <meta name="viewport" content="width=device-width">
    <meta charset="UTF-8">
    <link rel="stylesheet"
      href="vendor/bootstrap/dist/css/bootstrap.css">
    <link rel="stylesheet"
      href="vendor/bootstrap/dist/css/bootstrap-theme.css">
  </head>
  <body>
    <div class="container">
      <div class="jumbotron">
      <h1>Página não encontrada!<h1>
      </div>
      <h2 class="text-center">
        <a href="/">Voltar para a página principal</a>
      </h2>
    </div>
  </body>
</html>
```

Fig. 9.1: Página não encontrada

Reinicie o servidor e experimente acessar uma página que não existe e veja o resultado. Além de não darmos uma pista de qual tecnologia utilizamos no servidor, melhoramos a experiência do usuário.

A intenção deste capítulo não foi cobrir todos os aspectos de segurança de uma aplicação, mas pequenas alterações com grande impacto.

Nosso próximo passo será otimizar nossa aplicação para que o usuário possa ter a melhor experiência possível, porém não faremos isso manualmente. Utilizaremos um *task runner* que automatizará grande parte de nossa tarefa.

CZ£ –¶ 10

Grunt: automação de tarefas front-end

"Qualquer coisa boa que construamos acaba por nos construir"
– Jim Rohn

Em algum momento em sua carreira de desenvolvedor você já deve ter executado uma série de tarefas mesmo com o projeto pronto, como empacotá-lo para depois enviá-lo via □p.

Por exemplo, se você é um desenvolvedor front-end já deve ter se preocupado em mini□car e concatenar seus scripts, inclusive já deve ter experimentado o uso de algum pré-processador como **LESS**.

A qualidade e quantidade de passos variam de acordo com as necessidades do seu projeto e essas necessidades acabam gerando um □uxo (*work-*

flow) identi□cável e que muitas vezes é documentado para ser usado por toda a equipe.

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

10.1 Bu □-□□o□ Z□ G□□□□

Para solucionar problemas como esse, foram criadas no mercado ferramentas de construção (build) de projetos como Ant, Gradle e Maven, mas há uma que nasceu voltada especialmente para programadores front-end: o **Grunt**.

O Grunt é um *task runner* **totalmente feito em JavaScript** tornando-o atrativo no mundo front-end, inclusive ele possui um acervo com mais de 2.000 plugins suportando as mais diversas funcionalidades.

Instalação

O Grunt nada mais é do que um módulo do Node.js instalado através do npm:

```
npm install grunt@0.4 --save-dev
```

Repare que no comando anterior usamos `--save-dev` diferente do que estávamos acostumados a fazer. A diferença é que dentro do arquivo `package.json` o módulo será incluído dentro da chave `devDependencies`. É nesta chave que indicamos as dependências que só fazem sentido no ambiente de desenvolvimento e não de produção.

grunt-cli

Para o Grunt funcionar ainda precisamos instalar seu cliente em linha de comando, o `grunt-cli`:

```
npm install -g grunt-cli@0.1
```

É importante que você tenha permissão de administrador já que estamos instalando o módulo globalmente através do parâmetro `-g`. Este módulo nos permite chamar o comando `grunt` em qualquer local em nosso terminal.

Repare que não usamos o parâmetro --save-dev . Não nos importa qual versão do cliente de linha de comando estamos usando, apenas a versão do Grunt do projeto.

Instalar o Grunt por projeto permite sua atualização sem afetar os demais projetos, algo que não seria possível se fosse instalado globalmente.

10.2 O Z[illegible] G[illegible] [illegible]u .[illegible]

O arquivo Gruntfile.js é onde con□guramos tarefas (*tasks*). Nele, criamos um módulo do Node.js que espera receber como parâmetro uma instância do Grunt.

```
// contatooh/Gruntfile.js

module.exports = function(grunt) {

};
```

Mas quem chamará essa função passando esta referência?

O comando grunt

Isso é feito através do comando grunt no terminal. Este comando carregará nosso Gruntfile.js passando para o módulo uma instância do Grunt.

Antes de pensarmos em qualquer tarefa, precisamos garantir que todas elas sejam aplicadas em uma cópia da pasta contatooh de nosso projeto, garantindo a integridade dos arquivos originais.

Poderíamos até tentar fazer isso usando os comandos padrões do Grunt, ainda assim, precisaríamos varrer recursivamente a pasta que desejamos copiar e executar uma série de comandos fragmentados.

Para esta tarefa, tão corriqueira, podemos usar um plugin do Grunt que também é um módulo do Node.js.

10.3 I[][] Z[]Z[]o[] [][][][] [][][]u [][] [][][][]

O primeiro plugin do Grunt que utilizaremos será o grunt-contrib-copy (https://github.com/gruntjs/grunt-contrib-copy) :

```
npm install grunt-contrib-copy@0.7 --save-dev
```

Pronto, mas isso ainda não é su□ciente. Apesar de ele ter criado a pasta grunt-contrib-copy dentro da node_modules, ele ainda não é enxergado pelo Grunt; precisamos carregá-lo. O carregamento de plugins do Grunt é feito no Gruntfile.js através da função grunt.loadNpmTasks:

```
/* recebe o objeto grunt como parâmetro*/
module.exports = function(grunt) {

  grunt.loadNpmTasks('grunt-contrib-copy');
};
```

Agora que entendemos a estrutura básica do GruntFile.js, precisamos con□gurar nossas tarefas.

10.4 TZ[u [Z[

Plugins no Grunt são con□gurados através da função grunt.initConfig({ }):

```
module.exports = function(grunt) {

    grunt.initConfig({
          tarefa1: {
              /* configurações da tarefa1 */
           },
          tarefa2: {
             /* configurações do tarefa2 */
          }
  });

  grunt.loadNpmTasks('grunt-contrib-copy');
};
```

Repare que a função grunt.initConfig recebe como parâmetro um objeto JavaScript. Esse objeto tem como propriedades os nomes das tasks que desejamos con□gurar. Sabemos pela documentação do grunt-contrib-copy que a propriedade que representa a task deste plugin é **copy**. Assim, temos:

```
module.exports = function(grunt) {

  grunt.initConfig({
      copy: {} /* configurações da tarefa */
  });

  grunt.loadNpmTasks('grunt-contrib-copy');
};
```

Uma tarefa pode ter subtarefas, porém na documentação do Grunt são chamadas de *targets* (alvos). Veremos a seguir como de□nir alvos de tarefas.

10.5 TZ□□u □ □

Cada tarefa do Grunt pode conter alvos (*targets*) diferentes, que podem ser entendidos como subtarefas. Por exemplo, podemos querer copiar a pasta contatooh inteira ou apenas a pasta contatooh/public:

```
grunt.initConfig({
  copy: {
    aplicacaoInteira: {/* copia a pasta contatooh */},
    apenasPublic: {/* copia apenas a pasta contatooh/public */}
  }
});
```

No exemplo anterior, não con□guramos os targets da task copy, mas se já estivessem criados, poderíamos executar no terminal:

```
grunt copy
```

O comando anterior executaria a task copy e seus targets aplicacaoInteira e apenasPublic. Mas e se quiséssemos rodar

apenas um target específico? Podemos qualificar a task indicando qual target queremos executar:

```
grunt copy:aplicacaoInteira
```

Agora que entendemos o mecanismo de task e target, configuraremos nossa task copy:

```
// contatooh/Gruntfile.js

module.exports = function(grunt) {
  grunt.initConfig({
    copy: {
      project: {
        expand: true,
        cwd: '.',
        src: ['**', '!Gruntfile.js', '!package.json',
              '!public/bower.json'],
        dest: 'dist'
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-copy');
};
```

Vamos entender os parâmetros da task copy:

- **expand**: quando true ativa o mapeamento dinâmico. No lugar de definirmos o nome de cada arquivo e seu destino, indicamos o diretório de trabalho (cwd), a origem (src) e o destino (desc).
- **cwd**: diretório padrão (*current work directory*) no qual as demais propriedades se basearão. Em nosso caso, queremos a própria pasta que contém nosso script Grunt, por isso utilizamos '.'.
- **src**: array com os arquivos que devem ser copiados. Usamos o *globbing pattern* '**' para copiar todos os arquivos e diretórios. Desconsideramos alguns arquivos adicionando o prefixo ! em cada um deles.

- **dest**: pasta de destino. Em nosso caso, a pasta dist, que é criada caso não exista.

Agora que temos tudo configurado, já podemos rodar a task copy através do Grunt:

```
$ grunt copy
Running "copy:project" (copy) task
Created 756 directories, copied 2875 files

Done, without errors.
```

A pasta dist foi criada, sendo uma réplica da pasta contatooh.

10.6 A[Z[[[[Z[Z [Z[u [Z[

Tudo funciona, mas imagine o seguinte cenário: alguém apagou algum arquivo do projeto. Nossa cópia precisa refletir o projeto, logo, rodamos novamente o comando grunt copy. O problema é que ele conterá o arquivo que foi apagado do projeto original e isso é um problema.

A task clean

Precisamos apagar a pasta dist antes de rodarmos novamente nossa task copy. Para isso, existe o plugin **grunt-contrib-clean** (https://github.com/gruntjs/grunt-contrib-clean) . Sua instalação é feita através do comando:

```
npm install grunt-contrib-clean@0.6 --save-dev
```

Não podemos nos esquecer de registrá-lo em nosso arquivo:

```
grunt.loadNpmTasks('grunt-contrib-clean');
```

E, por fim, configurar a task:

```
module.exports = function(grunt) {
    grunt.initConfig({
        copy: {
            project: {
```

```
                expand: true,
                cwd: '.',
                src: ['**', '!Gruntfile.js', '!package.json',
                    '!public/bower.json'],
                dest: 'dist'
            }
        },
        clean: {
            dist: {
                src: 'dist'
            }
        }
    });

    grunt.loadNpmTasks('grunt-contrib-copy');
    grunt.loadNpmTasks('grunt-contrib-clean');
};
```

Usamos `clean` porque este é o nome da task fornecido por sua documentação. Escolhemos o nome `dist` como seu target para deixar claro que ele operará sobre a pasta `dist`, porém o que realmente define qual pasta será apagada é o objeto que o target recebe como parâmetro. Este objeto contém a chave `src`. Nela, definimos o diretório que será apagado. Testando a task `clean`:

```
$ grunt clean
Running "clean:dist" (clean) task
Cleaning dist...OK

Done, without errors.
```

Agora é só torcer para não esquecermos de executar as tasks na ordem correta:

```
grunt clean
grunt copy
```

Você ainda pode executar as duas de uma vez:

```
grunt clean copy
```

Queremos automatizar nossas tarefas, e termos que nos lembrar de executar ambas em ordem não se coaduna com nosso objetivo.

Para resolver problemas como esse, o Grunt permite registrar novas tasks que funcionam como uma espécie de atalho. Quando ela for chamada, automaticamente outras tasks serão chamadas na sequência que definirmos. Fazemos isso através da função grunt.registerTask:

```
grunt.registerTask('dist', ['clean', 'copy']);
```

Repare que essa função recebe como primeiro parâmetro o nome da nossa task. O segundo é um array com o nome das tasks já configuradas pelo Grunt. A ordem é importante, pois a primeira será executada antes da segunda e por aí vai. Nosso script final fica:

```
module.exports = function(grunt) {

  grunt.initConfig({
    copy: {
      project: {
        expand: true,
        cwd: '.',
        src: ['**', '!Gruntfile.js', '!package.json',
              '!public/bower.json'],
        dest: 'dist'
      }
    },

    clean: {
      dist: {
        src: 'dist'
      }
    }
});

grunt.registerTask('dist', ['clean', 'copy']);
grunt.loadNpmTasks('grunt-contrib-copy');
grunt.loadNpmTasks('grunt-contrib-clean');
};
```

Agora basta executarmos no terminal:

```
$ grunt dist
Running "clean:dist" (clean) task

Running "copy:project" (copy) task
Created 756 directories, copied 2875 files

Done, without errors.
```

O Grunt ainda permite registrarmos a task default. Ela será executada quando executarmos o comando grunt sem parâmetros. Modi□cando nosso script:

```
module.exports = function(grunt) {
  grunt.initConfig({
    copy: {
      project: {
        expand: true,
        cwd: '.',
        src: ['**', '!Gruntfile.js', '!package.json',
              '!public/bower.json'],
        dest: 'dist'
      }
    },

    clean: {
      dist: {
        src: 'dist'
      }
    }
  });

  grunt.registerTask('default', ['dist']);
  grunt.registerTask('dist', ['clean', 'copy']);
  grunt.loadNpmTasks('grunt-contrib-copy');
  grunt.loadNpmTasks('grunt-contrib-clean');
};
```

Basta executar no terminal:

```
$ grunt
Running "clean:dist" (clean) task
Cleaning dist...OK

Running "copy:public" (copy) task
Created 4 directories, copied 5 files

Done, without errors.
```

Agora que aprendemos o básico do Grunt, já podemos automatizar tarefas que podem melhorar a performance de nossa aplicação.

10.7 A[[vh[[hZ[ou [[[[[[hZl e[u h[[hZ[u [Zl e[

Uma prática muito comum em aplicações web é a concatenação e mini□cação de arquivos .css e .js.

A técnica de concatenação

A técnica de concatenação junta arquivos .css em um único arquivo, a mesma coisa com arquivos .js. A vantagem disso é que estaremos diminuindo o número de requisições de busca a esses recursos ao carregarmos a página. Quanto menor o número de requisições, mais rapidamente sua página será carregada, ainda mais se você estiver em uma rede do tipo 3G onde a latência domina.

A técnica de minificação

A mini□cação não visa atacar a latência, mas a largura de banda. Nela, tanto nos arquivos .css quanto nos .js removemos quebras de linha escrevendo tudo em uma única linha! Com isso, conseguimos economizar alguns bytes que, no somatório □nal, conseguem reduzir aproximadamente 10% do tamanho original do arquivo. Porém, os arquivos .js ainda permitem renomear o nome de funções e seus parâmetros para a menor quantidade possível de caracteres. Com certeza, o código □cará ilegível, mas carregará mais rápido na página do usuário.

Nunca altere os arquivos originais!

Um ponto importante é que tanto a concatenação quanto a minificação não devem ser feitas nos arquivos originais do projeto, caso contrário, a manutenção do sistema se tornará impossível. Precisamos realizar esse processo antes do deploy da aplicação, isto é, antes de ela entrar no ar.

Imagine realizar concatenação e minificação manualmente toda vez que alguma coisa mudar em nosso sistema? Já se imaginou concatenando arquivos css e ainda alterando a página index.ejs para apontar para o arquivo concatenado? Impraticável, não? É por isso que o Grunt possui uma série de plugins que podem nos ajudar a automatizar essa tarefa.

10.8 [illegible]

Minificação e merge de arquivos .js e .css com certeza são algumas das tarefas mais executadas no Grunt. Porém, não são tarefas que estão prontas por padrão. Precisamos instalar alguns plugins. Os plugins envolvidos nestas tarefas são:

- **grunt-contrib-concat**: https://github.com/gruntjs/grunt-contrib-concat concatena arquivos .css e .js.
- **grunt-contrib-uglify**: https://github.com/gruntjs/grunt-contrib-uglify minifica arquivos .js.
- **grunt-contrib-cssmin**: https://github.com/gruntjs/grunt-contrib-cssmin) minifica arquivos .css.

Como todo plugin, precisamos instalar cada um deles através do npm:

```
npm install grunt-contrib-concat@0.5 grunt-contrib-uglify@0.6
grunt-contrib-cssmin@0.10 --save-dev
```

E carregá-los no Gruntfile.js através da função grunt.loadNpmTasks:

```
// contatooh/Gruntfile.js

// código anterior omitido
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-cssmin');
```

O problema é que seremos responsáveis pela configuração das tasks de

O TIPO aceita dois valores: css e js. RESULTADOé o arquivo gerado e sua localização é sempre relativa à página. Em nossa página app/views/index.ejs, convencionaremos que o resultado da concatenação e minificação de arquivos CSS da página será css/index.min.css, e dos arquivos JavaScript, js/index.min.js. Alterando nossa página index.ejs, temos:

```
<!-- build:css css/index.min.css -->
<link rel="stylesheet"
  href="vendor/bootstrap/dist/css/bootstrap.css">
<link rel="stylesheet"
  href="vendor/bootstrap/dist/css/bootstrap-theme.
css">
<!-- endbuild -->

<!-- build:js js/index.min.js -->
<script src="vendor/angular/angular.js"></script>
<script src="vendor/angular-route/angular-route.js"></script>
<script src="vendor/angular-resource/angular-resource.js">
</script>
<script src="vendor/angular-i18n/angular-locale_pt-br.js">
</script>
<script src="js/main.js"></script>
<script src="js/controllers/ContatosController.js"></script>
<script src="js/controllers/ContatoController.js"></script>
<script src="js/services/ContatoService.js"></script>
<!-- endbuild -->
```

Não podemos nos esquecer das páginas auth.ejs e 404.ejs. Primeiro, vamos alterar auth.ejs:

```
<!-- build:css css/auth.min.css -->
<link rel="stylesheet"
  href="vendor/bootstrap/dist/css/bootstrap.css">
<link rel="stylesheet"
  href="vendor/bootstrap/dist/css/bootstrap-theme.css">
<!-- endbuild -->
```

Por fim, 404.ejs:

```
<!-- build:css css/404.min.css -->
<link rel="stylesheet"
  href="vendor/bootstrap/dist/css/bootstrap.css">
<link rel="stylesheet"
  href="vendor/bootstrap/dist/css/bootstrap-theme.css">
<!-- endbuild -->
```

Excelente! Agora, só nos resta configurar o grunt-usemin em nosso GruntFile.js.

10.9 A Z u u P u Z u

Para que a mágica do grunt-usemin aconteça, precisamos configurar duas tasks distintas. A primeira se chama useminPrepare. Ela gerará configurações dinâmicas para grunt-contrib-concat, grunt-contrib-uglify, grunt-contrib-cssmin, livrando-nos de configurá-los em nosso Gruntfile.js.

A segunda task se chama usemin. Ela alterará nossos arquivos HTML fazendo com que eles apontem para os arquivos concatenados e minificados definidos nos comentários especiais, pois foram criados antes pela task useminPrepare.

Não se preocupe, as duas tasks são tão simples quanto as que vimos anteriormente:

```
usemin : {
  html: 'dist/app/views/**/*.ejs'
},
useminPrepare: {
  html: 'dist/app/views/**/*.ejs'
}
```

Repare que ambas recebem como parâmetro um objeto que contém a chave html. É nele que indicamos quais páginas serão processadas; em nosso caso, todas as que estiverem dentro da pasta contatooh/dist/app/views/, evitando que as views originais em contatooh/app/views sejam modificadas. Porém, temos um problema.

Você lembra que os arquivos definidos nos metadados da página são criados relativamente à localização da página? Isso fará com que sejam criados indevidamente em contatooh/dist/app/views. Como eles não estão dentro da página pública, a página carregada no navegador não conseguirá baixá-los.

Podemos resolver isso indicando para a task useminPrepare, aquela que cria nossos arquivos, que crie os arquivos dentro da pasta contatooh/dist/public, passando um target que, na verdade, age como um objeto de configuração interna da task:

```
usemin : {
  html: 'dist/app/views/**/*.ejs'
},
useminPrepare: {
  options: {
    root: 'dist/public',
    dest: 'dist/public'
  },
  html: 'dist/app/views/**/*.ejs'
}
```

É importante frisar que não escolhemos qualquer nome para a task options. Vários plugins do Grunt utilizam essa convenção quando queremos alterar alguma configuração padrão da task. Nela, passamos um objeto com as chaves root e dest que possuem como valor o local onde queremos que os arquivos sejam gravados, em nosso caso, dist/public.

Regitrando task de atalho

Para que possamos concluir nosso script, precisamos garantir a execução ordenada das seguintes tasks:

1) useminPrepare: lê os metadados das páginas e cria as configurações para as tasks concat, uglify e cssmin.

2) concat: concatena os arquivos .js e .css utilizando como nome do arquivo a configuração gerada por useminPrepare.

3) uglify: minifica scripts com base na configuração gerada por useminPrepare.

4) cssmin: minifica arquivos css com base na configuração gerada por useminPrepare.

5) usemin: por fim, altera o HTML para que aponte para os arquivos concatenados e minificados.

Para isso, criaremos um atalho para todas elas chamado **minifica**:

```
grunt.registerTask('minifica', ['useminPrepare', 'concat',
  'uglify', 'cssmin', 'usemin']);
```

Por último, faremos com que nossa task *default*, além de criar uma cópia do projeto, chame automaticamente nossa task:

```
grunt.registerTask('default', ['dist', 'minifica']);
```

Nosso script final deverá estar assim:

```
module.exports = function(grunt) {
  grunt.initConfig({
    copy: {
      project: {
        expand: true,
        cwd: '.',
        src: ['**', '!Gruntfile.js', '!package.json',
              '!public/bower.json'],
        dest: 'dist'
      }
    },

    clean: {
      dist: {
        src: 'dist'
      }
    },

    usemin : {
```

```
            html: 'dist/app/views/**/*.ejs'
        },

        useminPrepare: {
            options: {
                root: 'dist/public',
                dest: 'dist/public'
            },
            html: 'dist/app/views/**/*.ejs'
        }

    });

    grunt.registerTask('default', ['dist', 'minifica']);
    grunt.registerTask('dist', ['clean', 'copy']);
    grunt.registerTask('minifica', ['useminPrepare', 'concat',
    'uglify', 'cssmin', 'usemin']);

    grunt.loadNpmTasks('grunt-contrib-copy');
    grunt.loadNpmTasks('grunt-contrib-clean');
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.loadNpmTasks('grunt-contrib-cssmin');
    grunt.loadNpmTasks('grunt-usemin');
};
```

Vamos realizar um teste completo. Rode nossa task default e depois suba o projeto dentro da pasta contatooh/dist. Se o seu script estiver correto, será exibida uma série de informações no terminal, inclusive os parâmetros de configuração gerados pelo grunt-usemin.

Edite a página contatooh/dist/app/views/index.ejs e verifique se a task usemin conseguiu realizar a substituição das tags e se foram gerados os arquivos contatooh/dist/public/css/index.min.css e contatooh/dist/public/js/index.min.css. Visualize cada um deles. Se você não conseguir entender o que eles exibem é um indício de que o processo de minificação funcionou.

Mas será que nosso sistema ainda funciona depois de tanta alteração? Entre na pasta contatoo/dist e suba nosso servidor:

```
node server
```

Autentique-se logo em seguida. Tudo funciona? Não! Assim que nos autenticamos, nada é exibido no navegador. Por quê? Se abrirmos o console do navegador, veremos a mensagem de erro:

```
Uncaught Error: [$injector:modulerr] Failed to instantiate
module contatooh due to:
Error: [$injector:unpr] Unknown provider: a
```

O problema é que o processo de minificação de scripts alterou o nome dos parâmetros de várias funções. Como o sistema de injeção de dependências do AngularJS baseia-se no nomes dos parâmetros, tudo deixou de funcionar.

Temos duas saídas. Podemos anotar nossas dependências, algo que deixará nosso código mais verboso. Por exemplo, nosso ContatoController ficaria assim:

```
angular.module('contatooh').controller('ContatoController',
  ['$scope', '$routeParams', 'Contato',
  function($scope, $routeParams, Contato) {
    // código do controller
  }]);
```

Repare que substituímos o segundo parâmetro da função controller por um array. Neste array, os três primeiros valores equivalem aos parâmetros da função, último parâmetro passado. Porém, como são dados, não serão alterados durante o processo de minificação. O AngularJS passa a se basear nesses dados para saber qual artefatos injetar dentro da função. Por mais que seus parâmetros sejam trocados pelo processo de minificação, teremos a certeza de que o processo de injeção funcionará como esperado.

Outra solução é deixar o código do jeito que está e automatizar essa tarefa para ser executada dentro da pasta contatooh/dist. É o que veremos na próxima seção.

10.10 A[illegible]u [illegible]hZle[illegible]

O **grunt-ng-annotate** (https://github.com/mzgol/grunt-ng-annotate) é um plugin do Grunt capaz de anotar injeções de dependência do AngularJS em

nosso código automaticamente. Sua instalação é feita via npm:

```
npm install grunt-ng-annotate@0.5 --save-dev
```

A task ngAnnotate altera nossos arquivos adicionando a sintaxe que vimos na seção anterior e preparando nossos arquivos para o processo de minificação. Vamos configurá-la e registrá-la em nosso Gruntfile.js:

```
// contatooh/Gruntfile.js

// código anterior omitido

ngAnnotate: {
    scripts: {
      expand: true,
      src: ['dist/public/js/**/*.js']
    },
}
// código posterior omitido

grunt.loadNpmTasks('grunt-ng-annotate');
```

Não esqueça de adicionar a task na lista de da nossa task minifica, mas antes de uglify:

```
grunt.registerTask('minifica', ['useminPrepare', 'ngAnnotate',
'concat', 'uglify', 'cssmin', 'usemin']);
```

Pronto! Rode o comando grunt no terminal. No console, será exibida a mensagem:

```
Running "ngAnnotate:scripts" (ngAnnotate) task
>> 4 files successfully generated.
```

Por fim, experimente entrar na pasta contatooh/dist e subir a aplicação. Tudo deve continuar funcionando.

CZ£ –¶ 11

Testando a aplicação

"A beleza das coisas existe no espírito de quem as contempla."
– David Hume

Uma ferramenta de teste de código é um programa que utiliza nosso código e compara o resultado de uma execução com um valor, ou estado de um objeto, que esperamos que seja devolvido ou alterado.

Muitas linguagens de programação contam com ferramentas que auxiliam este processo. Uma das ferramentas para teste de JavaScript mais populares do mercado é o **Jasmine**. Esta ferramenta pode ser usada tanto para testes de unidade quanto para testes de ponta a ponta:

- **Teste de unidade**: no inglês, *unit testing*. Foca em uma coisa apenas, por exemplo, um controller de nosso sistema. Não há dependência externa como o acesso a banco de dados ou serviços.

- **Teste de ponta a ponta**: no inglês, *end-to-end testing (e2e)*. Testa se o □uxo do requisito de uma aplicação do ponto de vista do usuário está funcionando como projetado do início ao □m.

Porém, a equipe do AngularJS não estava satisfeita com o tempo entre a alteração de uma funcionalidade do sistema e a execução de seu teste de unidade. Eles queriam que alterações feitas em qualquer funcionalidade do sistema disparassem automaticamente seu teste de unidade e fornecessem um feedback imediato para desenvolvedor. Esta foi a motivação da criação do **Karma**.

Não satisfeitos, eles também criaram o **Protractor**, uma ferramenta específ□ca para testes e2e em aplicações com AngularJS que facilita enormemente a utilização do Selenium. Neste capítulo, abordaremos essas tecnologias e como integrá-las à nossa aplicação.

11.1 KZ□□Z: □u u ofZh□ □□□□Z□□□□u □ ou □u □□ u □ ou □□□□bZou

Karma (http://karma-runner.github.io/) é um ambiente de testes focado na produtividade do desenvolvedor livrando-o de seus detalhes de con□guração. A ideia é que o desenvolvedor tenha um feedback instantâneo do resultado de seus testes de unidade quando qualquer JavaScript da aplicação for alterado. O Karma possui outras características que o tornam ainda mais interessante:

- **Testes em dispositivos reais**: navegadores como Chrome, Firefox entre outros. Suporte ao PhamtonJS (http://phantomjs.org/) , um navegador *headless* baseado no Webkit (https://www.webkit.org/) , que roda sem interface visual. Tablets e smartphones entram na lista.
- **Agnóstico do framework de teste**: você pode usar Jasmine (http://jasmine.github.io/) , Mocha (http://visionmedia.github.io/mocha/) , QUnit (http://qunitjs.com/) ou criar um adapter para seu framework de teste favorito.
- **Integração descomplicada**: integração com Jenkins (http://jenkins-ci.org/) , Travis (https://travis-ci.org/) ou Semaphore

(https://semaphoreapp.com/) .

- **Para todos**: não é exclusivo do AngularJS, podendo ser utilizado por qualquer aplicação que queira automatizar seus testes e obter o feedback de seus testes de unidade em tempo real.

Instalação

O ambiente criado pelo Karma roda sob o Node.js e sua instalação é feita pelo npm:

```
npm install karma@0.12 --save-dev
```

Precisamos também de seu cliente de linha de comando instalado globalmente (privilégio de administrador necessário) para podermos iniciá-lo em qualquer diretório de nosso sistema de arquivos.

```
npm install -g karma-cli@0.1
```

Com o Karma instalado, já podemos configurá-lo.

karma.config.js

Apesar de o Karma nos fornecer um ambiente para executarmos nossos testes de unidade, precisamos configurar seu arquivo karma.config.js. Nele, informamos a localização de nossos scripts e bibliotecas, o framework de teste escolhido, os navegadores utilizados para teste entre outras configurações.

No lugar de criarmos o arquivo manualmente, podemos pedir ao seu assistente que crie um para nós. Vamos criar o arquivo dentro da pasta contatooh/config, porém rodaremos o comando dentro da pasta raiz contatooh:

```
$ karma init config/karma.config.js
```

O sistema fará uma série de perguntas, mas apenas duas merecem destaque neste momento. Para o restante, daremos ENTERaté terminar o assistente:

```
Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next
question.
> jasmine
```

Utilizaremos o Jasmine como framework de teste, opção exibida por padrão pelo Karma.

```
Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move
to the next question.
> Chrome
```

Precisamos indicar qual navegador utilizaremos para rodar os testes, inclusive podemos indicar mais de um. Em nosso caso, □caremos apenas com o Chrome por enquanto. Mais tarde, trabalharemos com o PhamtomJS, um browser sem interface visual muito utilizado em servidores de integração continua, assunto que abordaremos no próximo capítulo.

Quando o processo terminar, dentro da pasta node_modules serão adicionados os módulos karma-jasmine e karma-chrome-launcher.

O Karma de nada servirá se não elaborarmos nossos testes no framework de teste escolhido. É por isso que aprenderemos como o Jasmine funciona na próxima seção.

11.2 JZ□□□□u : □□Z□u□□□ ou □u□□u

O Jasmine (http://jasmine.github.io/) é um framework de testes voltado para JavaScript que não depende de um navegador para funcionar, muito menos do DOM. Ele possui um conjunto de funções que nos auxiliam na realização de testes.

Nosso primeiro candidato para testes será public/js/controllers/ContatoController.js. Por convenção, escrevemos testes em um arquivo de mesmo nome acrescido do su□xo Spec, logo, teremos o arquivo ContatoControllerSpec.js. Todos nossos testes □carão dentro da pasta contatooh/test/spec. Com o arquivo contatooh/test/spec/ContatoControllerSpec.js criado, já podemos dar início à nossa primeira suíte de testes.

11.3 C□□Z□o□ □□□□ u □ ou □ u □□ u □

O Jasmine nos permite criar suítes de testes através da função describe disponibilizada pelo próprio framework. A função recebe dois parâmetros: o primeiro é a descrição de nossa suíte e o segundo é uma função que conterá uma série de testes que no Jasmine são chamados de **Specs**:

```
// contatooh/test/spec/ContatoControllerSpec.js

describe("ContatoController", function() {

});
```

O nome da suíte é tipicamente o nome da classe, função ou componente que desejamos testar. É o nome da suíte que será exibido com o resultado de seus testes no terminal. Nosso próximo passo será de□nir os comportamentos de ContatoController que desejamos testar.

Criando Specs através da função it

Dentro de nossa suíte, criaremos uma a□rmativa, nossa Spec, sobre nosso objeto ContatoController através da função it:

```
describe("ContatoController", function() {

    it("Deve criar um Contato vazio quando
        nenhum parâmetro de rota for passado", function() {

    });
});
```

O primeiro parâmetro, nossa a□rmação, deixa claro para quem está lendo o código o que estamos esperando, mas é o comportamento de□nido na função passada como segundo parâmetro que testará essa **expectativa**. É este comportamento que precisamos de□nir.

A função expect

O Jasmine usa a função expect para testar a expectativa de nossa Spec:

```
describe("ContatoController", function() {

    it("Deve criar um Contato vazio quando
        nenhum parâmetro de rota for passado", function() {

        expect($scope.contato._id).toBeUndefined();
    });
});
```

Repare que, nesse exemplo, a função expect recebe como parâmetro o _id do contato presente no escopo de ContatoController. Em seguida, encadeamos a chamada da função toBeUndefined, isto é, para que nosso teste passe, o _id do contato deve ser undefined. Sabemos que isso deve acontecer quando nenhum parâmetro de rota for passado para o controller, exatamente a descrição da função it.

> Para uma lista exaustiva de todas as funções suportadas por expect, acesse o link http://jasmine.github.io/2.0/introduction.html.

Mas espere um pouco! Como $scope apareceu em nosso código? Jasmine saberá instanciar ContatoController? Antes de nos preocuparmos se o teste passará ou não, queremos que ele seja executado pelo Karma.

11.4 R□o□Z□o□□□□ u □ou □ u □□ u □Z□ □Z□□ o□ KZ□□Z

Para que o Karma seja capaz de executar nossos testes criados pelo Jasmine, precisamos editar seu arquivo contatooh/config/karma.config.js e adicionar na chave files o caminho ../test/spec/ ** / * Spec.js.

```
files: ['../test/spec/**/*Spec.js']
```

Descemos um nível com .. porque o diretório padrão que o Karma utilizará quando o rodarmos será o mesmo diretório de seu arquivo

```
sChrome 37.0.2062:
Executed 1 of 1 (1 FAILED) ERROR (0.011 secs / 0.008 secs)
```

As primeiras informações do terminal dizem respeito ao servidor Karma criado em http://localhost:9876/ que roda eternamente, razão do travamento de seu terminal. Ainda no terminal, temos informações como a versão do navegador utilizado, entre outras, mas o que nos interessa agora é saber se nossa suíte de testes foi processada e, realmente, ela foi. É fácil perceber que sua única Spec não passou no teste, pois consta o estado FAILED.

Outro ponto importante é o referenceError. Nele, temos a mensagem "gscope is not de□ned", ou seja, o Jasmine não foi capaz de resolver sozinho o escopo do controlador que queremos testar e, por isso, a exceção invalidou nosso teste.

Feedback constante

Precisamos fechar o terminal agora? Não! A ideia é termos o Karma rodando o tempo todo enquanto criamos nossos testes. Qualquer alteração dos arquivos de□nidos na chave files de karma.config.js disparará novos testes automaticamente. Isso permitirá que o desenvolvedor receba um feedback o mais rápido possível caso algum teste falhe!

Mas a pergunta que você deve estar fazendo é: como conseguiremos instanciar ContatoController em nossa Spec, uma vez que nosso controller somente é criado através do AngularJS? A resposta está na combinação do Jasmine com o módulo **Angular-Mocks**.

11.5 A□□□Z□-□h□□u □□u □□Zl e□h□ JZ□□□u

Como o AngularJS utiliza pesadamente injeção de dependências, podemos solicitar diretamente de seu sistema de injeção quais artefatos precisamos em nossas Specs, tudo através de fun ções especiais disponibilizadas pelo módulo Angular-Mocks. Também podemos trocar, quando necessário, esses artefatos por objetos falsos (*mocks*) que muitas vezes nos ajudam na criação de nossos testes. Veremos isso tudo na prática.

Instalamos o módulo angular-mocks através do bower executando o comando a seguir dentro da pasta contatooh:

```
bower install angular-mocks#1.3 --save-dev
```

O próximo passo é fazer com que o Jasmine carregue o módulo para que possamos utilizá-lo. Aproveitaremos para adicionar o caminho de todos os arquivos utilizados pelo projeto. Alterando nosso arquivo contatooh/config/karma.config.js:

```
files: [
    '../public/vendor/angular/angular.js',
    '../public/vendor/angular-mocks/angular-mocks.js',
    '../public/vendor/angular-resource/angular-resource.js',
    '../public/vendor/angular-route/angular-route.js',
    '../public/js/main.js',
    '../public/js/controllers/**/*.js',
    '../public/js/services/**/*.js',
    '../test/spec/**/*Spec.js'
],
```

A□□ u □Zl e□ o□ □Z□□Z.h□□□□.□□

Toda vez que o arquivo karma.config.js for modi□cado, você precisará reiniciar o Karma.

Excelente, agora já podemos refatorar nossa Spec.

A função module

Sabemos que é através da diretiva ng-app da nossa página principal que o AngularJS carrega o módulo contatooh de□nido no arquivo contatooh/public/main.js, todavia precisamos carregá-lo manualmente.

O Angular-Mocks possui a função module que recebe como parâmetro o nome do módulo que queremos carregar. Contudo, em que lugar de nossa suíte de testes carregaremos o módulo contatooh?

A função beforeEach

O Jasmine possui a função beforeEach sempre chamada antes da execução de cada uma de nossas Specs. Parece ser um bom lugar:

```
describe("ContatoController", function() {

    beforeEach(function() {
        module('contatooh');
    });

    it("Deve criar um Contato vazio quando
        nenhum parâmetro de rota for passado", function() {

        expect($scope.contato._id).toBeUndefined();
    });
});
```

Carregamos através da função module o módulo principal de nossa aplicação, mas como teremos acesso ao ContatoController dentro de nossa Spec?

A função inject

Para que possamos acessar controllers dentro de nossos testes, substituiremos o callback da função it por uma chamada à função inject, disponibilizada pelo Angular-Mocks.

```
describe("ContatoController", function() {

    beforeEach(function() {
        module('contatooh');
    });

    it("Deve criar um Contato vazio quando
        nenhum parâmetro de rota for passado",
        inject(function() {
            expect($scope.contato._id).toBeUndefined();
    }));
});
```

vamos inicializá-la com um código especial dentro da função beforeEach que já criamos:

```
describe("ContatoController", function() {

    var $scope;

    beforeEach(function() {
        module('contatooh');
        inject(function($injector) {
            $scope = $injector.get('$rootScope').$new();
        });
    });

    it("Deve criar um Contato vazio quando
        nenhum parâmetro de rota for passado",
        inject(function($controller) {

            $controller('ContatoController');
            expect($scope.contato._id).toBeUndefined();
    }));
});
```

Recorremos mais uma vez à função inject, mas não podemos simplesmente solicitar a injeção de $scope porque ele só é injetado em controllers. Não podemos apenas criar um objeto qualquer com papel de escopo, pois o escopo de todo controller herda de $rootScope equivalente ao elemento onde colocamos a diretiva ng-app. É aí que entra o serviço $injector (não o confunda com a função inject).

O serviço $injector permite instanciar outros artefatos do AngularJS que não sejam serviços. É através de $injector.get('$rootScope') que temos acesso à definição de $rootScope, mas estamos interessados em uma instância desta definição que criamos encadeando uma chamada à função $injector.get('$rootScope').$new(). Conseguimos criar um $scope novinho em folha!

Criamos nosso $scope, mas como o associaremos ao ContatoController criado em nossa Spec? O serviço controller

recebe, além do nome do controller que queremos instanciar um segundo parâmetro, um objeto cuja chaves representam os artefatos injetados que desejamos substituir. Sendo assim, temos:

```
describe("ContatoController", function() {

    var $scope;

    beforeEach(function() {
        module('contatooh');
        inject(function($injector) {
            $scope = $injector.get('$rootScope').$new();
        });
    });

    it("Deve criar um Contato vazio quando
        nenhum parâmetro de rota for passado",
        inject(function($controller) {

            $controller('ContatoController',
                        {"$scope" : $scope});
            expect($scope.contato._id).toBeUndefined();
    }));
});
```

Pronto! Nosso teste instanciou ContatoController e nossa função expect tem acesso ao seu $scope. Sabemos que durante a inicializaç ão do controller é veri□cado se buscamos um contato ou instanciamos um novo. Como em nosso teste não entramos no mérito deste parâmetro, um novo contato instanciado deve ter seu _id undefined. Não se lembra da lógica? Você pode consultá-la na seção 5.11.

Se você não terminou a execução do Karma, assim que você salvar as alterações verá instantaneamente o resultado do teste no terminal. Aliás, o teste deve passar:

```
Chrome 37.0.2062: Executed 1 of 1 SUCCESS (0 secs / 0.035 secs)
Chrome 37.0.2062: Executed 1 of 1 SUCCESS (0.037 secs / 0.035
secs)
```

Agora que nossa primeira Spec passou, já podemos ir um pouco além criando um teste mais so□sticado na próxima seção.

11.6 g□□ □ □BZh□u □o: “□□h□Z□o□” □□□□□ f Zh□u □o

Na seção anterior, aprendemos a instanciar um controller em nossos testes, inclusive criamos e associamos um escopo a ele. Este escopo foi utilizado pela função expec do Jasmine para testar nossa expectativa do que deveria acontecer.

Criaremos uma nova Spec através de outra chamada à função it. Essa Spec veri□cará a condição contrária da Spec que criamos: *deve preencher o Contato quando parâmetro de rota for passado*.

Desta vez, quando instanciarmos ContatoController, além de passarmos o escopo que criamos para o controller, passamos um objeto que representará nosso $routeParams, aquele artefato do AngularJS que consultamos em nosso controller para saber se algum parâmetro foi passado. A diferença é que estipularemos o valor 1 para ele.

```
describe("ContatoController", function() {
    var $scope

    // código anterior omitido

    it("Deve preencher o Contato quando
        parâmetro de rota for passado",
        inject(function($controller) {
            $controller('ContatoController', {
                '$routeParams': {contatoId: 1},
                '$scope': $scope
            });
            expect($scope.contato._id).toBeDefined();
    }));
});
```

Assim que você salvar, verá no terminal (se você não tiver fechado o Karma) que nosso teste não passa. Recebemos a exceção:

```
"TypeError: Cannot read property '_id' of undefined"
```

Isso acontece porque, quando passamos o contatoId, nosso controller tentará recuperar o contato através de uma requisição para nosso REST Endpoint. Como não estamos rodando nosso servidor, o callback de erro de ContatoController será chamado, desta maneira, $scope.contato será undefined. Repare que o importante aqui é testar se nossa lógica funciona, não realizar uma conexão real com o banco.

Para resolver problemas como esse, o Angular-Mocks possui o $httpBackend para mockarmos nosso back-end. Isso mesmo! Podemos simular a resposta para qualquer um de nossos REST Endpoints. A ideia é retornarmos um contato com _id igual ao que estipulamos em nosso $routeParams.

Definindo as respostas de nossos Endpoints

Precisamos pedir uma instância de $httpBackEnd através da função inject. Faremos isso dentro da função beforeEach do Jasmine:

```
describe("ContatoController", function() {
    var $scope, $httpBackend;

    beforeEach(function() {
        module('contatooh');
        inject(function($injector, _$httpBackend_){
            $scope = $injector.get('$rootScope').$new();
            $httpBackend = _$httpBackend_;
        });
    });
});
```

Declaramos a variável $httpBackend, que será acessível dentro de nossas Specs, porém você deve ter reparado que, dentro da função inject, injetamos o serviço _$httpBackend_. Usamos *underline* no nome do serviço para diferenciá-lo da variável $httpBackend. Internamente, o sistema de injeção removerá os *underlines* para que nosso código funcione. Se não tivéssemos feito isso, teríamos problemas em associar o $httpBackend recebido como parâmetro com a variável declarada de mesmo nome.

A pergunta que você deve estar se fazendo é o motivo de não termos ado-

tado outro nome de variável no lugar de $httpBackend, removendo assim a necessidade dos *underlines* do parâmetro injetado. A razão é simples: queremos usar os mesmos nomes dos artefatos do AngularJS em nossos testes. Foi por isso que declaramos no teste anterior $scope como variável que armazena o escopo do controller e não outro nome.

Agora já podemos preparar as respostas para **quando** a rota /contato/1 for acessada.

ghttpBackend.when

Preparamos as respostas de nossos EndPoints através da função $httBackend.when:

```
describe("ContatoController", function() {
    var $scope, $httpBackend;

    beforeEach(function() {
        module('contatooh');
        inject(function($injector, _$httpBackend_){
            $scope = $injector.get('$rootScope').$new();
            $httpBackend = _$httpBackend_;
            $httpBackend.when('GET', '/contatos/1')
                                .respond({_id: '1'});
        });
    });

    // código posterior omitido
});
```

A função $httBackend.when recebe como parâmetro a URI do Endpoint acessado. Em seguida, encadeamos uma chamada à função respond, que devolverá o contato criado por nós. Repare que estamos devolvendo apenas um objeto com a chave _id, suficiente para atender nosso teste.

Pronto! Será que nosso teste passa? Não! Parece que nossa solução ainda não esta completa, pois continuamos com o mesmo erro:

```
TypeError: Cannot read property '_id' of undefined
```

Parece que nosso back-end de mentirinha não funcionou, mas por quê?

ghttpBackend.flush

O motivo de nosso teste ter falhado é pelo fato de $httpBackend trabalhar sincronamente. Como assim? Repare que a chamada de nossa função expect vem depois da criação do controller. Se $httpBackend fosse assíncrono, nossa função expect não funcionaria. Precisamos indicar em que ponto de nosso teste queremos que as requisições sejam resolvidas de uma só vez. Fazemos isso chamando $httpBackend.flush:

```
describe("ContatoController", function() {
    var $scope, $httpBackend;

    it("Deve preencher o Contato quando
        parâmetro de rota for passado",
        inject(function($controller) {
            $controller('ContatoController', {
                $routeParams: {contatoId: 1},
                '$scope': $scope
            });
            $httpBackend.flush();
            expect($scope.contato._id).toBeDefined();
    }));
});
```

Com o Karma rodando, você receberá outra mensagem de erro:

```
Error: Unexpected request: GET /contatos
No more request expected
```

Qual o motivo dessa vez?

Mocke apenas Endpoints acessados durante o teste

Nosso flush funcionou, porém quando usamos $httpBackend precisamos mockar os Endpoints chamados pelo controller. Não precisamos mockar todos os Endpoints, apenas aqueles acessados de acordo com o

critério de nossa Spec. Difícil se lembrar deles? Não se preocupe, você receberá uma mensagem de erro para cada um até que prepare uma resposta para todos eles.

> O Karma pode ser usado em duas fases. A primeira, enquanto desenvolvemos para obtermos feedback em tempo real, e a segunda, através de um servidor de integração, algo que veremos nos próximos capítulos.

Vamos recordar que ContatoController carrega em sua inicialização uma lista de contatos (esta é uma boa hora de rever o código na seção 5.6). Sendo assim, vamos mockar a resposta para a URI /contatos retornando uma lista com apenas um objeto, inclusive ele não terá chave alguma. Não se preocupe, isso é suficiente já que não temos interesse na lista de contatos, apenas em resolver a expectativa da requisição em nosso teste.

Por fim, nosso teste ficará assim:

```
describe("ContatoController", function() {
    var $scope, $httpBackend;

    beforeEach(function() {
        module('contatooh');
        inject(function($injector, _$httpBackend_){
            $scope = $injector.get('$rootScope').$new();
            $httpBackend = _$httpBackend_;
            $httpBackend.when('GET', '/contatos/1')
                        .respond({_id: '1'});
            $httpBackend.when('GET', '/contatos')
                        .respond([{}]);
        });
    });

    it("Deve criar um Contato vazio quando
        nenhum parâmetro de rota for passado",
        inject(function($controller) {
            $controller('ContatoController', {
                '$scope': $scope
            });
```

```
            expect($scope.contato._id).toBeUndefined();
    }));

    it("Deve preencher o Contato quando
        parâmetro de rota for passado",
        inject(function($controller) {
            $controller('ContatoController', {
                $routeParams: {contatoId: 1},
                '$scope': $scope
            });
            $httpBackend.flush();
            expect($scope.contato._id).toBeDefined();
    }));
});
```

Até aqui, neste capítulo, vimos como realizar testes de unidade com Jasmine, Karma e Angular-Mocks, mas ainda falta aprendermos como realizar testes de ponta a ponta, os famosos **e2e tests** (*end to end test*).

11.7 Tu Z o ou Z Z Z

Na seção anterior, testamos pequenas partes de nosso sistema através de testes de unidade. Mas será que testar apenas partes isoladas de nosso sistema é su ciente? Talvez uma analogia possa nos dar uma resposta.

Uma metáfora, um problema

Imagine que você tenha criado um quebra-cabeça daqueles que fragmentam a foto de um cenário surreal em zilhões de partes. O acabamento de cada peça é meticulosamente testado para garantir a melhor qualidade para o cliente nal. Porém, uma semana após o lançamento, alguns clientes devolveram o produto alegando que algumas peças faltavam ou que algumas delas não se encaixavam ou então se encaixavam com muito folga, prejudicando o resultado nal.

Se pudesse voltar ao passado, o que você poderia ter feito para evitar essa catástrofe? Com certeza devem ter vindo à sua cabeça mais de uma resposta, mas o que acha de ter montado cada quebra-cabeça antes de embalá-lo ver-

i□cando o resultado □nal? Deslocar funcionários para esta □nalidade com certeza seria algo custoso, inclusive o processo não seria um dos mais rápidos.

Como testar sua aplicação como um todo?

E se o quebra-cabeça fosse nossa aplicação Contatooh? Como testá-la como um todo, isto é, não no nível de unidade, mas seu funcionamento recebendo dados, interagindo com o servidor web e gravando no banco de dados? Será que um roteiro com todos os passos que devem ser testados é su□ciente? Quem executará esses passos, você ou sua equipe de qualidade? E se um erro acontecer? Ele será reportado em tempo hábil para não acontecer o deploy de uma aplicação com problemas?

Automatizando, automatizando, automatizando

Aprendemos a automatizar tarefas com Grunt, inclusive a automatizar testes de unidade com Karma. Será que não podemos fazer a mesma coisa com nossos testes de ponta a ponta? Imagine se no lugar do roteiro tivéssemos um script executável que simulasse a interação homem-máquina e que procurasse por resultados previsíveis. Com certeza, saberíamos quando algo inesperado acontecesse, inclusive teríamos a certeza de que nenhum passo seria pulado. Resumindo: **queremos automatizar nossos testes** end-to-end **(e2e)**.

Testes e2e com Selenium

Testes *end-to-end* (e2e) procuram garantir que os componentes integrados de uma funcionalidade da aplicação funcionam como esperado. Toda a aplicação é testada em um cenário do mundo real, tais como a comunicação com o banco de dados, rede, hardware e outras aplicações.

O **Selenium** (http://www.seleniumhq.org) é a ferramenta *de facto* para automatizar testes e2e em navegadores. Toda mágica de automação ocorre através do seu **WebDriver** escrito para diversos navegadores do mercado.

Os testes do Selenium são escritos na linguagem Java, mas isso não impede que outras linguagens sejam utilizadas. Por exemplo, há o projeto **Web-**

DriverJs (http://webdriver.io/) criado para o Node.js.

11.8 P□□□ □Zh□ □□: □ u □□ u □ u □o-□ □-u □o h□□A□□□-□Z□JS

Nossa tarefa nesta seção será criar testes e2e da nossa aplicação Contatooh, no entanto, há um problema que precisamos atacar.

Imagine que você tenha automatizado o clique no botão salvar da página de cadastro de contatos. Sabemos que o texto "Salvo com sucesso" aparecerá caso o contato tenha sido salvo e nada mais justo do que testarmos por sua presença para sabermos se ele foi realmente incluído. O problema é que o teste será realizado antes do término da requisição assíncrona que grava o contato, o que invalidaria nosso teste.

Uma solução é adicionar temporizadores em nossos testes antes de cada veri□cação, tornando o processo de criação do teste ainda mais complexo. Foi pensando neste problema e em outros que a equipe do AngularJS criou um framework para testes e2e chamado **Protractor**.

Protractor (http://angular.github.io/protractor) é um framework de testes end-to-end para aplicações feitas com AngularJS. Ele é construído sob o WebDriver para interagir com sua aplicação como um usuário real faria. Em outras palavras, Protractor é uma casquinha que envolve o WebDriver, permitindo que nossos testes sejam executados sem nos preocuparmos com temporizadores toda vez que testarmos nossa aplicação. Há também funções exclusivas que nos ajudam a interagir com vários recursos do AngularJS.

Instalação

Instalamos o Protractor globalmente (privilégio de administrador necessário) através do npm:

```
npm install -g protractor@1.5
```

Esse comando, além de instalar o Protractor, se encarregará da instalação do WebDriver que deve ser atualizado logo em seguida pelo comando:

```
webdriver-manager update
```

Para subir o Selenium Server:

```
webdriver-manager start
```

Esse comando inicializará o Selenium Server, que receberá requisições de todos os nossos testes feitos com Protractor. O servidor controlará seu navegador localmente, inclusive podemos acessar seu status no endereço (http://localhost:4444/wd/hub) .

11.9 Conﬁgurando o Protractor para a aplicação

Para que o Protractor funcione, precisamos criar um arquivo de conﬁguração que chamaremos de protractor.js dentro da pasta config. É nele que indicamos o caminho de nossas Specs, neste caso, especíﬁcas para testes e2e:

```
// config/protractor.js

exports.config = {
  specs: ['../test/e2e/**/*.js']
};
```

Já temos o Selenium Server rodando e o Protractor conﬁgurado. Criaremos os scripts contatosPageSpec.js e contatoPageSpec.js dentro da pasta contatooh/test/e2e. O primeiro conterá os testes para a view contatos.html, e o segundo, para contato.html. Será que falta mais alguma coisa antes de criarmos nossas Specs e2e?

11.10 Automatizando o processo de autenticação

Precisamos automatizar o processo de autenticação do usuário antes da execução de nossos testes. Para essa ﬁnalidade, podemos adicionar no arquivo de conﬁguração do Protractor a chave onPrepare. Essa chave recebe como valor uma função que será chamada apenas uma vez antes da execução de

nossas suítes de testes. É um bom começo para aprendermos como o Protractor interage com o Selenium.

> É recomendada a criação de uma conta no Github exclusiva para esta finalidade, mas nada impede que você utilize sua mesma conta (tomando cuidado para não divulgar sua senha).

O primeiro passo é solicitar ao navegador que abra a página http://localhost:3000. Fazemos isso através da função browser.get, mas, como nossa página auth.ejs e a página de login do Github não utilizam AngularJS, teremos problemas: Protractor espera eternamente o carregamento do AngularJS, o que nunca irá acontecer nessas páginas, o que trava nossos testes. Lembre-se que o Protractor é um *wrapper* em torno de uma instância do WebDriver, sendo assim, podemos interagir diretamente com este último através de browser.driver:

```
exports.config = {
  specs: ['../test/e2e/**/*Spec.js'],
  onPrepare: function() {
    browser.driver.get('http://localhost:3000');
  }
};
```

Sabemos que para usuários não autenticados o sistema exibirá a view auth.ejs com o link "Entre pelo Github". Precisamos ter uma referência para este link em nosso teste para que possamos clicá-lo. Fazemos isso através da função findElement:

```
exports.config = {
  specs: ['../test/e2e/**/*.js'],
  onPrepare: function() {
    browser.driver.get('http://localhost:3000');
    browser.driver.findElement(by.id('entrar')).click();
  }
};
```

Vamos entender como essa função funciona e como ela trabalha com outros objetos globais.

Du ZfZo h l e h f u u S h

Existe uma outra forma de evitar que o Protractor busque pelo AngularJS em páginas que não o utilizem, evitando a necessidade de interagir com o WebDriver diretamente. Porém, ela não funcionou corretamente com a versão 1.4 do Protractor:

```
browser.ignoreSynchronization = true;
// comandos
browser.ignoreSynchronization = false;
```

O autor sugere que esta forma seja evitada até que a equipe do Protractor tenha alguma posição sobre seu não funcionamento.

A função findElement

A função `findElement` nada mais é do que uma função utilitária para achar e interagir com elementos do DOM da página que estamos testando. Ela recebe como único parâmetro um **locator strategy**.

Locator Strategy

Obtemos as estratégias de localização através do objeto `by`. Em nosso caso, procuramos o elemento pelo seu ID `entrar` (tenha certeza de tê-lo digitado quando criou a página `auth.ejs`). Em seguida, para o elemento retornado, solicitamos um clique.

Sabemos que o usuário será direcionado para a página de login do GitHub. Nesta página, precisamos interagir com três elementos: os inputs do login e senha e o botão que efetua o login. Inspecionando a página, sabemos facilmente que os inputs possuem os IDs `login_field` e `password` respectivamente.

A função sendKeys

Preencheremos os dois inputs através da função `sendKeys`:

```
// config/protractor.js

exports.config = {
  specs: ['../test/e2e/**/*.js'],
  onPrepare: function() {
    browser.driver.get('http://localhost:3000');
    browser.driver.findElement(by.id('entrar')).click();
    browser.driver.findElement(by.id('login_field'))
        .sendKeys('email-de-teste');
    browser.driver.findElement(by.id('password'))
        .sendKeys('senha-do-email-de-teste');
  }
};
```

Agora só nos resta clicar no botão para efetuarmos o login, porém ele não possui um id, apenas o atributo name com o valor commit. Podemos selecionar o elemento através da *locator strategy* by.name:

```
// config/protractor.js

exports.config = {
  specs: ['../test/e2e/**/*.js'],
  onPrepare: function() {
    browser.driver.get('http://localhost:3000');
    browser.driver.findElement(by.id('entrar')).click();
    browser.driver.findElement(by.id('login_field'))
        .sendKeys('email-de-teste');
    browser.driver.findElement(by.id('password'))
        .sendKeys('senha-do-email-de-teste');
    browser.driver.findElement(by.name('commit')).click();
  }
};
```

Ainda não criamos nossas Specs, mas isso não nos impede de rodar o Protractor e verificar se a autenticação está sendo realizada. Só verifique se você criou os arquivos contatosPageSpec.js e contatoPageSpec.js mesmo vazios dentro da pasta contatooh/test/e2e, caso contrário, o Protractor exibirá uma mensagem de erro alegando que nenhum arquivo foi

encontrado dentro da pasta contatooh/test/e2e. Rodamos o Protractor através do comando:

```
protractor config/protractor.js
```

Quando rodarmos nosso teste, o navegador será aberto, o link, clicado, e a página de login do GitHub preenchida automaticamente. No console, teremos a informação:

```
Finished in 0 seconds
0 tests, 0 assertions, 0 failures
```

Mas de nada isso serve se não testarmos cenários chaves de nossa aplicação.

11.11 Tu Z o hu

Nossa próxima tarefa será criar uma suíte de testes específica para a página principal http://localhost:3000/k/contatos. Para essa suíte, criaremos uma Spec que verificará se o usuário está logado. Utilizaremos a já conhecida função beforeEach do Jasmine para garantir que estamos na página correta antes de cada uma de nossas specs. Vamos editar o arquivo test/e2e/contatosPageSpec.js:

```
// test/e2e/contatosPageSpec.js

describe('Página principal', function() {

    beforeEach(function() {
        browser.get('http://localhost:3000/#/contatos');
    });
});
```

Repare que dessa vez utilizamos diretamente browser e não browser.driver como fizemos anteriormente. Agora faz todo sentido utilizamos o *wrapper* do WebDriver disponibilizado pelo Protractor porque estamos testando uma página que possui o AngularJS carregado.

Agora criaremos nossa primeira Spec através da função it:

```
// test/e2e/contatosPageSpec.js

describe('Página principal', function() {

    beforeEach(function() {
        browser.get('http://localhost:3000/#/contatos');
    });

    it('Deve estar logado', function() {
        element(by.id('usuario-logado')).getText()
        .then(function(texto) {
            expect(texto.trim().length).toBeGreaterThan(0);
        });
    });
});
```

Repare que, no lugar de usarmos a função findElement, utilizamos element, esta última também fornecida pelo Protractor. Sua vantagem é que, além das locator strategies que já aprendemos, ela aceita outras especí□cas do AngularJS.

Nesse código, a primeira coisa que fazemos em nossa Spec é buscar o elemento com o ID usuario-logado, nossa tag span da view index.ejs. De posse do elemento, chamamos a função getText, porém ela não retorna o texto do elemento, mas uma *promise*. Isso acontece pela natureza assíncrona dos testes com Protractor. Sabemos que toda *promise* possui a função then e nela temos acesso ao texto do elemento. Por □m, usamos a função expect do Jasmine que vimos na seção 11.2. Ela testa se o tamanho do texto do elemento é maior que zero. Rodando mais uma vez nosso teste:

```
protractor config/protractor.js
```

No console, veri□camos que nosso teste passou:

```
Finished in 8.475 seconds
1 test, 1 assertion, 0 failures
```

Testando outros cenários

Que tal agora criarmos um teste para nosso cadastro de contatos? Vamos editar o arquivo test/e2e/contatoPageSpec.js:

```
describe('Cadastro de contatos', function() {

    beforeEach(function() {
        browser.get('http://localhost:3000/#/contato');
    });

    it('Deve cadastrar um contato', function() {

        var aleatorio =
            Math.floor((Math.random() * 10000000) + 1);
        var nome = 'teste' + aleatorio;
        var email = 'teste@email' + aleatorio;
    });
});
```

Repare que aqui abrimos a tela de cadastro e, logo em seguida, criamos dados aleatórios para o contato. Precisamos de uma referência para os inputs que capturam o nome e o e-mail do contato, mas desta vez não utilizaremos a função by.id nem by.name.

by.model: locator Strategy exclusivo do Protractor

O Protractor disponibiliza a função by.model, que permite selecionar um elemento pelo valor de sua diretiva ng-model:

```
// test/e2e/contatoPageSpec.js

describe('Cadastro de contatos', function() {

    beforeEach(function() {
        browser.get('http://localhost:3000/#/contato');
    });

    it('Deve cadastrar um contato', function() {
```

```
        var aleatorio =
            Math.floor((Math.random() * 10000000) + 1);
        var nome = 'teste' + aleatorio;
        var email = 'teste@email' + aleatorio;
        element(by.model('contato.nome')).sendKeys(nome);
        element(by.model('contato.email')).sendKeys(email);
    });
});
```

Preenchemos o nome e o e-mail, ainda falta selecionar uma emergência da lista. Os valores da diretiva ng-options usada para criar nossa lista de emergências sempre vão de "0" até o tamanho da lista menos um. Por debaixo dos panos, o que o Angular considera mesmo é o valor do model. Desta vez, procuraremos o elemento por um seletor de atributo através da função by.css:

```
// test/e2e/contatoPageSpec.js

describe('Cadastro de contatos', function () {

    beforeEach(function() {
        browser.get('http://localhost:3000/#/contato');
    });

    it('Deve cadastrar um contato', function() {

        var aleatorio =
            Math.floor((Math.random() * 10000000) + 1);
        var nome = 'teste' + aleatorio;
        var email = 'teste@email' + aleatorio;
        element(by.model('contato.nome')).sendKeys(nome);
        element(by.model('contato.email')).sendKeys(email);
            element(by.css('option[value="0"]')).click();
    });
});
```

Pronto! Já temos os campos preenchidos, precisamos agora salvar o contato:

```
// test/e2e/contatoPageSpec.js

describe('Cadastro de contatos', function() {

    beforeEach(function() {
        browser.get('http://localhost:3000/#/contato');
    });

    it('Deve cadastrar um contato', function() {

        var aleatorio =
            Math.floor((Math.random() * 10000000) + 1);
        var nome = 'teste' + aleatorio;
        var email = 'teste@email' + aleatorio;
        element(by.model('contato.nome')).sendKeys(nome);
        element(by.model('contato.email')).sendKeys(email);
            element(by.css('option[value="0"]')).click();
        element(by.css('.btn-primary')).click();
    });
});
```

Por fim, precisamos verificar se a mensagem de sucesso foi exibida para termos certeza do salvamento do contato. Lembre-se que exibimos a mensagem através de uma *Angular Expression* (seção 5.11). O Protractor permite selecionar um elemento através de sua AE pela função by.binding:

```
// test/e2e/contatoPageSpec.js

describe('Cadastro de contatos', function () {

    beforeEach(function() {
        browser.get('http://localhost:3000/#/contato');
    });

    it('Deve cadastrar um contato', function() {

        var aleatorio =
            Math.floor((Math.random() * 10000000) + 1);
        var nome = 'teste' + aleatorio;
```

```
        var email = 'teste@email' + aleatorio;
        element(by.model('contato.nome')).sendKeys(nome);
        element(by.model('contato.email')).sendKeys(email);
        element(by.css('option[value="0"]')).click();
        element(by.css('.btn-primary')).click();
        expect(element(by.binding('mensagem.texto'))
            .getText())
            .toContain('sucesso');
    });
});
```

Depois do texto encontrado, verificamos se ele contém o texto "sucesso". Se contiver, nosso teste passará. Está na dúvida? Rode mais uma vez nossos os testes:

```
protractor config/protractor.js
```

Voltaremos para o arquivo test/e2e/contatosPageSpec.js e, nele, adicionaremos mais uma Spec que testará a remoção de um contato de nossa lista.

Para sabermos se um contato foi removido, precisamos do total da lista antes e depois da remoção comparando-os no final. Para isso, utilizaremos a função by.repeat que recebe como parâmetro o valor da diretiva ng-repeat. Os elementos criados dinamicamente pela ng-repeat também ganham a diretiva com o mesmo valor:

```
// test/e2e/contatosPageSpec.js

describe('Página principal', function () {

    // código anterior omitido

    it('Deve remover um contato da lista', function() {

        // by repeater é coisa do protractor
        var totalAntes = element
            .all(by.repeater('contato in contatos'))
            .count();
```

```
    });
});
```

Agora que já temos o total de contatos, precisamos remover um deles clicando no botão "Remover". Como encontraremos esse botão? Podemos usar a função by.css, passando uma classe como parâmetro, mas teremos um problema: o botão "Remover" é repetido para cada item da lista, logo, teremos elementos diferentes com a mesma classe. Uma solução é pegar a primeira linha da lista e, a partir dela, procurar o elemento com a classe btn:

```
// test/e2e/contatosPageSpec.js

describe('Página principal', function() {

    // código anterior omitido

    it('Deve remover um contato da lista', function() {

        var totalAntes = element
            .all(by.repeater('contato in contatos'))
            .count();

        element(by.repeater('contato in contatos').row(0))
            .element(by.css('.btn'))
            .click();
    });
});
```

Excelente! Agora, precisamos contar o total de contatos da lista após a exclusão para, em seguida, testar nossa expectativa:

```
// test/e2e/contatosPageSpec.js

describe('Página principal', function() {

    // código anterior omitido

    it('Deve remover um contato da lista', function() {
```

```
        var totalAntes = element
            .all(by.repeater('contato in contatos'))
            .count();

        element(by.repeater('contato in contatos').row(0))
            .element(by.css('.btn'))
            .click();
        var totalDepois = element
            .all(by.repeater('contato in contatos')).count();
        expect(totalDepois).toBeLessThan(totalAntes);
    });
});
```

É só rodar mais uma vez nossos testes:

```
protractor config/protractor.js
```

E verificar o resultado:

```
Finished in 12.822 seconds
3 tests, 3 assertions, 0 failures
```

> Para que nosso teste funcione, precisamos ter no mínimo um contato cadastrado, caso contrário, não seremos capazes de escolher um contato na lista de contatos de emergência. Não se preocupe, na próxima seção aprenderemos como popular um banco específico para testes antes de rodarmos nossos testes e2e.

Executamos duas suítes de testes com um total de três Specs, três asserções e nenhuma falha.

Legebilidade dos testes

Aprendemos a criar testes e2e com Protractor que nada mais é do que uma casquinha em volta do WebDriver do Selenium que traz novas estratégias de localização de elementos, inclusive nos remove a responsabilidade de trabalharmos com *timeouts* toda vez que formos testar alguma ação assíncrona.

Mas será que nossos testes são legíveis? E se eles tivessem o triplo de interações do usuário? E se os elementos da página mudassem (tag, IDs, classes etc.)? Talvez este seja um forte argumento contra testes e2e: manutenção e legibilidade.

Na próxima seção, utilizaremos um padrão de projeto que nos ajudará na manutenção e legibilidade de nossos testes e2e.

11.12 PZ□u Of□uh□: □u □□□□Z□o□ Z □u □f□□□bZou u □Z□□□ u □l e□ou □ u □□ u □

O padrão de projeto **PageObject** (http://martinfowler.com/bliki/PageObject.html) procura esconder os detalhes da interface do usuário de outros componentes do sistema, em nosso caso, nossos scripts de teste. Ele consegue adicionando em um objeto funções de alto nível como obterMensagem, digitarNome, salvar, que delegam seu trabalho para as funções de manipulação de DOM, como as do Protractor que utilizamos. Se o ID, classe ou qualquer outra propriedade de um elemento da interface do usuário mudar, alteramos apenas em seu PageObject e nosso teste continuará intacto.

Vamos criar a pasta test/e2e/pages e convencionar que todos os nossos PageObjects □carão nessa pasta. Em seguida, criaremos o arquivo test/e2e/pages/contatoPage.js, nosso PageObject para a página contato.html:

```
// test/e2e/pages/contatoPage.js

var contatoPage = function() {

    this.visitar = function() {
        browser.get('http://localhost:3000/#/contato');
    };

    this.digitarNome = function(nome) {
        element(by.model('contato.nome')).sendKeys(nome);
    };

    this.digitarEmail = function(email) {
```

```
            element(by.model('contato.email')).sendKeys(email);
    };

    this.salvar = function() {
        element(by.css('.btn-primary')).click();
    };

    this.obterMensagem = function() {
        return element(by.binding('mensagem.texto')).getText()
    };

    this.selecionarPrimeiraEmergenciaDaLista = function() {
        element(by.css('option[value="0"]')).click();
    };
}
module.exports = contatoPage;
```

Veja que criamos um módulo do Node.js que devolve uma função construtora. Agora vamos refatorar test/e2e/contatoPageSpec.js para usar nosso PageObject.

```
// test/e2e/contatoPagesSpec.js

var ContatoPage = new require('./pages/contatoPage');

describe('Cadastro de contatos', function () {
    var pagina = new ContatoPage();

    beforeEach(function() {
        pagina.visitar();
    });

    it('Deve cadastrar um contato', function() {

        var aleatorio =
            Math.floor((Math.random() * 10000000) + 1);
        pagina.digitarNome('teste' + aleatorio);
        pagina.digitarEmail('teste@email' + aleatorio);
        pagina.selecionarPrimeiraEmergenciaDaLista();
```

```
        pagina.salvar();
        expect(pagina.obterMensagem()).toContain('sucesso');
    });
});
```

Você consegue agora enxergar em nosso teste alguma referência às funções de manipulação de DOM do Protractor? Ficou mais fácil de ler? E se em contato.html o botão salvar mudasse sua classe de btn-primary para btn-default? Nosso teste falharia, com certeza, porém corrigiríamos esse problema alterando o PageObject, deixando nosso teste intacto. Vamos rodar nosso teste mais uma vez e verificar se tudo continua funcionando:

```
protractor config/protractor.js
```

Que tal agora criarmos um PageObject para test/e2e/contatosPageSpec.js? Vamos criar o arquivo test/e2e/pages/contatosPage.js:

```
// test/e2e/pages/contatosPage.js

var contatosPage = function() {

    this.visitar = function() {
        browser.get('http://localhost:3000/#/contatos');
    };

    this.obterUsuarioLogado = function(nome) {
        return element(by.id('usuario-logado')).getText();
    };

    this.obterTotalDeItensDaLista = function() {

        return element.all(by.repeater('contato in contatos'))
                      .count();
    };

    this.removerPrimeiroItemDaLista = function() {

        element(by.repeater('contato in contatos').row(0))
```

```
                .element(by.css('.btn'))
                .click();
    }
}
module.exports = contatosPage;
```

Por fim, vamos refatorar test/e2e/contatosPageSpec.js utilizando nosso PageObject:

```
// test/e2e/contatosPageSpec.js

var ContatosPage = new require('./pages/contatosPage');

describe('Página principal', function() {

    var pagina = new ContatosPage();

     beforeEach(function() {
         pagina.visitar();
     });

    it('Deve estar logado', function() {

         pagina.obterUsuarioLogado().then(function(texto) {
             expect(texto.trim().length).toBeGreaterThan(0);
         });
      });

    it('Deve remover um contato da lista', function() {

         var totalAntes = pagina.obterTotalDeItensDaLista();
         pagina.removerPrimeiroItemDaLista();
         var totalDepois = pagina.obterTotalDeItensDaLista();
         expect(totalDepois).toBeLessThan(totalAntes);
    });
});
```

A diferença da legibilidade desta última alteração ficou ainda mais evidente! Agora é só rodar nossos testes:

```
protractor  config/protractor.js
```

Tudo deve continuar funcionando, pois só refatoramos nosso código.

CZ£ –¶ 12

Integração continua

"Nunca salte de um trampolim quebrado"
– William Shakespeare

Criamos a aplicação Contatooh passo a passo e é muito provável que você tenha sido o único desenvolvedor neste processo. Porém, no mundo de desenvolvimento de so□ware, é extremamente comum termos equipes dos mais diversos tamanhos trabalhando em um mesmo projeto. Já imaginou integrar o trabalho de cada desenvolvedor de cada equipe? Por mais que usemos um sistema de controle de versão como o Git, Subversion ou Mercurial não estaremos livres de problemas de integração.

No capítulo anterior, criamos testes de unidade e e2e para garantirmos qualidade de nosso so□ware. E que tal se nossos testes fossem executados a cada integração de código? A ideia é boa, mas primeiro precisaríamos realizar a construção (*build*) de nossa aplicação a cada integração, inclusive torcer

para não esquecermos de executar nossos testes. Muita responsabilidade para o desenvolvedor, não? Independente da dificuldade, o que queremos realizar é a **integração contínua** de nossa aplicação para recebermos feedback o mais rápido possível sobre possíveis erros de integração.

Servidores de integração

Para solucionar problemas como esse, foram criados servidores especializados na construção e execução de testes disparados a cada integração do código da aplicação. Eles são chamados de **servidores de integração**. No mercado existem vários servidores com essa finalidade, como o Jenkins/Hudson, Semaphore, CruiseControl, entre outros. Neste capítulo, aprendemos a realizar integração contínua utilizando o **Travis**.

12.1 T□Z□□ CI

Travis CI (https://travis-ci.com/) é um servidor de integração continua como serviço. Ele é integrado com o GitHub e suporta várias plataformas como Node.js e Java. Você pode utilizar gratuitamente a versão open source (https://travis-ci.org/) ou optar pela versão paga (https://magnum.travis-ci.com) , que permite build concorrente entre outras vantagens.

O primeiro passo para utilizarmos o Travis é termos uma conta no Github (https://github.com) e um repositório exclusivo para nossa aplicação Contatooh. Não nos aprofundaremos nos detalhes do Git, focaremos apenas nos passos que devem ser feitos para sua integração com o Travis.

C□□□ □□□Z□o□ □□□u □h□□ G□□ uG□□ H□f

Não domina Git ainda ou quer saber mais? Você encontra na própria Casa do Código o livro *Controlando versões com Git e GitHub*, de Alexandre Aquiles e Rodrigo Ferreira.

Preparando o repositório

No Github, criaremos o repositório **contatooh**. Antes de associarmos o

repositório com nosso projeto, crie o arquivo .gitignore dentro da pasta local contatooh. Este arquivo serve para indicarmos de quais arquivos ou pastas não queremos controlar versão. Nele, adicionaremos:

```
.tmp
.DS_Store
.DS_Store?
node_modules/
public/vendor/
dist/
script/
```

Em seguida, vamos associá-lo à pasta local contatooh, nossa aplicação, através dos comandos:

```
git init
git add .
git commit -m "primeiro commit"
git remote add origin git@github.com:suaContaNoGit/contatooh.git
git push -u origin master
```

Excelente! Nosso código já está disponibilizado no repositório público criado em seu Github. Nossa próxima tarefa será configurar nosso repositório para se integrar com o Travis.

12.2 C□□□□□□Z□o□ .□ □Z□□.□□

Para que possamos integrar nosso repositório com o Travis, precisamos criar seu arquivo .travis.yml. Este arquivo está no formato YAML(*YAML Ain't Markup Language*, http://www.yaml.org/) , voltado para serialização de dados legíveis por humanos. Não entraremos nos detalhes dessa linguagem, porém é importante respeitarmos a indentação do arquivo, caso contrário, ele terá uma estrutura inválida.

> Existe um utilitário de configuração disponibilizado pelo Travis feito em Ruby, porém o autor preferiu não utilizá-lo nesta etapa do projeto para que o leitor entenda mais detalhadamente o arquivo .travis.yml e sua importância. Mais tarde, ele será utilizado no capítulo 14.5.

Crie o arquivo .travis.yml dentro da pasta contatooh. Nosso primeiro passo será definir a linguagem e a versão que serão utilizadas pelo Travis, inclusive um e-mail de notificação que receberá mensagens sobre o status do build:

```
language: node_js
node_js:
  - 0.10
notifications:
  - seuemail@email.com
```

É claro que essas configurações ainda não são suficientes. Precisamos adicionar mais configurações.

Ciclo de Vida

O Travis possui um *ciclo de vida* e sua primeira fase é chamada before_install. Nela, baixamos todos arquivos de que precisamos antes de o Travis automaticamente chamar o comando npm install para nós. Em nosso caso, instalaremos alguns módulos chaves globalmente, assim como fizemos em nossa máquina local. Também solicitaremos ao bower que baixe nossas dependências front-end já que o Travis não o chama automaticamente:

```
before_install:
  - npm install -g bower@1.3
  - npm install -g grunt-cli@0.1
  - npm install -g karma-cli@0.0
  - npm install -g protractor@1.5
  - webdriver-manager update
  - bower install
```

Imediatamente após a fase before_install, o Travis baixará todas nossas dependências através do comando npm install. Isso não é suficiente, pois precisamos rodar nosso script do Grunt que minifica e concatena nossos scripts. É por isso que existe a fase before_script. Ela é anterior à fase que executa nossos testes:

```
before_script:
  - grunt
  - cd dist
```

Como instalamos o Grunt globalmente na base before_install, o comando grunt funcionará gerando a pasta dist. Em seguida, mudamos o diretório de trabalho para esta pasta. O motivo é simples: queremos rodar nossos testes com o projeto devidamente minificado e concatenado, versão que mais tarde entrará em produção.

Agora queremos executar nossos testes unitários através do Karma, mas não usando o Chrome ou Firefox. Para que nosso teste rode rapidamente, utilizaremos o PhantomJS, um navegador sem interface gráfica que pode ser chamado por linha de comando. Para isso, precisamos instalar seu lançador especialmente feito para o Karma:

```
npm install karma-phantomjs-launcher@0.1 --save-dev
```

> Não esqueça que novos arquivos precisam ser adicionados com o comando git add e que qualquer alteração feita precisa ser sincronizada com o repositório através do comando git push.

Agora, na fase script chamaremos o Karma, mas indicando que ele utilizará o PhantomJS, e passaremos o parâmetro --single-run para que ele execute os testes e finalize logo em seguida. Isso é necessário para que o Karma não trave o nosso servidor de integração, aliás, só faz sentido monitorar arquivos no ambiente de desenvolvimento, não de testes.

```
script:
 - karma start config/karma.config.js --single-run --browsers
 PhantomJS
```

Excelente! Isso já é suficiente para construirmos e rodarmos nossos testes de unidade. Precisamos agora criar uma conta no Travis.

12.3 A□□□h□Z□o□ □u □□u □□□□□ □□□□ Z□ T□Z□□

Criar uma conta gratuita no Travis é extremamente simples. Basta acessar o endereço (https://travis-ci.org) e, no canto direito superior, clicar no link "Sign in Github". Na primeira vez, será exibida uma mensagem para que você con□rme que o Travis poderá ter acesso ao seu repositório. Advinhem que protocolo ele está utilizando? O OAuth 2.0, aquele mesmo protocolo que utilizamos para a autenticação de nosso sistema na seção 8.1.

Depois de con□rmar, no canto esquerdo da tela aparecerá uma aba com o título My repositories. Ela exibe todos os repositórios monitorados pelo Travis, mas, como acabamos de criar nossa conta, nenhum é exibido. Adicionamos novos repositórios através do botão com sinal de soma:

Fig. 12.1: Repositórios monitorados e adição de novos repositórios

Será exibida uma página que listará todos os seus repositórios de sua conta no GitHub. Caso nenhum seja exibido, basta clicar no botão sync now. Para cada um, haverá um botão de on/o□. Precisamos ligá-lo para o repositório contatooh. Porém, o Travis só realizará o monitoramento caso haja o arquivo .travis.yml na raiz do repositório.

Fig. 12.2: Liga/desliga monitoramento do repositório pelo Travis

Depois de ativar o repositório, ao voltarmos para a página (http://

acessar o Travis, mas será que existe alguma maneira de o nosso repositório nos fornecer esta informação? Sim! Foi pensando nisso que o Travis criou uma barra de status que pode ser adicionada no README.mdde nosso repositório do GitHub:

Fig. 12.5: Repositório no Github exibindo status do build

Para adicionar o status exibido na imagem anterior, basta acessar seu repositório no Travis e clicar em build Unknown:

Fig. 12.6: Status do build / tipos de notificações

Quando clicado, será exibido um diálogo no qual poderemos escolher de uma lista o formato *Markdown* utilizado pelo README.mddo GitHub:

Fig. 12.7: Markdown de status de build

Agora é só copiar e colocar o conteúdo exibido no README.mddo repositório de nossa aplicação. Qualquer um que acessar o repositório terá

uma forma rápida de saber em que situação encontra-se a construção do projeto.

E nossos testes e2e com Protractor?

Poderíamos pedir para que o Travis rodasse esses testes para nós, mas além do tempo de espera, que não seria um dos melhores, caríamos limitados a utilizar o PhamtonJS ou o Firefox. Para não termos limites no que podemos conseguir em nossos testes e2e, utilizaremos em conjunto com o Travis um serviço especializado para rodar testes deste tipo, assunto da próxima seção.

12.4 TZZS: u Z Z u hu h SZhu LZf

No lugar de nos preocuparmos com a infraestrutura na qual nossos testes e2e rodarão (conguração do Selenium Server, suporte a todos os navegadores do mercado sejam eles Desktop ou mobile etc.) podemos rodar nossos testes utilizando serviços de terceiros com esta nalidade, os famosos TaaS (*test as a service*).

Sauce Labs (http://saucelabs.com) é um TaaS gratuito para projetos open source e pago para projetos fechados. Ele é facilmente integrado com diversos servidores de integração contínua do mercado, inclusive o Travis que vimos na seção anterior.

Su Z Z

O servidor gratuito do Sauce Labs oferece:

- Duas horas de testes automatizados;
- Todos os navegadores de Desktop e Mobile;
- Duas máquinas virtuais concorrentes.

O Sauce Labs, além de rodar nossos testes nos navegadores que desejarmos, ainda tira uma screenshot de cada ação de nosso script de teste que pode ser consultado, inclusive ele cria um screencast composto por todas as screenshots para que possamos visualizar como o teste foi realizado. O primeiro passo para utilizarmos o Sauce Labs é realizarmos nosso registro em seu site (https://saucelabs.com/signup) .

Depois do registro, precisamos preparar nosso sistema para adotar parâmetros de con□guração diferentes de acordo com o ambiente em que nossa aplicação é executada.

12.5 P□u □Z□Z□o□ Z Z□□□hZl e□ □Z□Z o□□u □u □□ u □ Z□f□u □□ u □

No ambiente de testes, queremos utilizar o banco contatooh_test no lugar de contatooh, inclusive queremos populá-lo com alguns contatos antes de cada teste. Também precisamos disponibilizar para o Travis os CLIENT_ID e CLIENT_SECRETda nossa aplicação, assim como o usuário e senha que utilizaremos em nossos testes e2e. Não podemos simplesmente deixar essas informações em nosso repositório público correndo o risco de elas vazarem. A ideia é criarmos variáveis de ambiente diferentes para o ambiente de desenvolvimento e testes.

O objeto global process

O núcleo de nossa solução mora no objeto global do Node.js chamado **process**. É através de process.env que temos acesso a qualquer variável de ambiente do sistema operacional. Um padrão muito utilizado para identi□car em qual ambiente nossa aplicação está rodando é o uso da variável NODE_ENV com os valores production, development e test. Vejamos um exemplo:

```
// apenas exemplo, não entra em nenhum .js
var clientID, clientSecret;

if (process.env.ENV == "test") {
  clientID = process.env.CLIENT_ID,
  clientSecret = process.env.CLIENT_SECRET
```

```
}

if (process.env.ENV == "development") {
  clientID = "XXX",
  clientSecret = "ZZZ"
}
// usa as variáveis clientID e clientSecret
```

Esse código identifica em qual ambiente estamos para saber quais valores adotar para o clientID e o clientSecret da aplicação. O problema desta solução é que teremos muitos IFs espalhados pelo nosso código.

Isolando as configurações de ambiente

Para resolvermos este problema, isolaremos as configurações de cada ambiente em um módulo em separado. Dentro da pasta contatooh/app/config/env, criaremos os arquivos development.js e test.js. Ainda não nos preocuparemos com as configurações de produção. Também não se preocupe com as variáveis que ainda não vimos, elas serão elucidadas ainda neste capítulo:

```
// app/config/env/test.js
module.exports = {
  env: 'test',
  db: 'mongodb://localhost/contatooh_test',
  sauceTestName: 'Contatooh E2E Testing',
  sauceUser : process.env.SAUCE_USERNAME,
  sauceKey : process.env.SAUCE_ACCESS_KEY,
  travisJobNumber: process.env.TRAVIS_JOB_NUMBER,
  travisBuild: process.env.TRAVIS_BUILD_NUMBER,
  clientID: process.env.CLIENT_ID,
  clientSecret: process.env.CLIENT_SECRET,
  seleniumUser: process.env.SELENIUM_USER,
  seleniumUserPassword: process.env.SELENIUM_USER_PASSWORD
};

// app/config/env/development.js

module.exports = {
```

```
  env: 'development',
  db: 'mongodb://localhost/contatooh',
  clientID: process.env.CLIENT_ID,
  clientSecret: process.env.CLIENT_SECRET,
  seleniumUser: process.env.SELENIUM_USER,
  seleniumUserPassword: process.env.SELENIUM_USER_PASSWORD
};
```

Vamos criar um módulo responsável pelo retorno da configuração de acordo com o valor da variável process.env.NODE_ENV. Para isso, crie o arquivo contatooh/app/config/config.js:

```
// app/config/config.js

module.exports = function() {
  return require('./env/' + process.env.NODE_ENV+ '.js');
}
```

Vejamos um exemplo de seu uso:

```
// exemplo apenas, não entra em nenhum js
var config = require('./config')();
console.log(config.seleniumUser);
console.log(config.seleniumUserPassword);
console.log(config.env);
```

Refatorando nossa aplicação para utilizar nosso objeto config

Agora precisamos refatorar nosso sistema para que ainda sejamos capazes de rodar nosso teste de integração, porém utilizando nosso objeto de configuração. Vamos começar alterando o arquivo contatooh/app/config/protractor.js:

```
var config = require('./config')();

// código anterior comentado

browser.driver.findElement(by.id('login_field'))
  .sendKeys(config.seleniumUser);
browser.driver.findElement(by.id('password'))
```

```
    .sendKeys(config.seleniumUserPassword);

// código posterior comentado
```

Agora, nosso arquivo contatooh/server.js:

```
var config = require('./config/config')();
// código anterior comentado
require('./config/database')(config.db);
// código posterior comentado
```

Não adicionamos CLIENT_ID nem o CLIENT_SECRET como variáveis de ambiente à toa. Precisamos alterar nosso arquivo app/config/passport.js. Nele, temos fixos os valores de clientID e clientSecret. Agora vamos lê-los de nosso objeto config:

```
// app/config/passport.js
var config = require('./config')();

// código comentado
clientID: config.clientID,
clientSecret: config.clientSecret,
// código comentado
```

Se estivermos em ambiente de desenvolvimento, nosso banco será contatooh, mas se estivermos em ambiente de testes, ele será contatooh_test. Será que já podemos subir nossa aplicação? Ainda não.

Script para inicialização das variáveis de ambiente

No mínimo, precisamos configurar as variáveis de ambiente utilizadas por config/env/development.js.

Vamos criar dois scripts que atribuirão as variáveis de ambientes de que precisamos. O primeiro será responsável pela inicialização de nosso servidor e se chamará contatooh/script/server.sh:

```
export NODE_ENV=development
export SELENIUM_USER=nome_de_um_usuario_no_github
export SELENIUM_USER_PASSWORD=senha_do_usuario
```

```
export CLIENT_ID=client_id_da_apliacao
export CLIENT_SECRET=senha_do_client
node server
```

O segundo, contatooh/script/test.sh, criará as variáveis de ambiente utilizadas pelo Protractor:

```
export NODE_ENV=development
export SELENIUM_USER=nome_de_um_usuario_no_github
export SELENIUM_USER_PASSWORD=senha_do_usuario
export CLIENT_ID=client_id_da_apliacao
export CLIENT_SECRET=senha_do_client
protractor config/protractor.js
```

> Poderíamos ter atribuído essas variáveis diretamente no sistema operacional, fazendo com que fi cassem sempre disponíveis, mas o autor preferiu declará-las em scripts para que o leitor não perca o foco da aplicação estudando detalhes do sistema operacional utilizado.

É muito importante não controlarmos as versões dos arquivos server.sh e test.sh. Queremos esses arquivos apenas localmente para que tenhamos uma maneira rápida e eficaz de atribuirmos as variáveis de ambiente e rodarmos nossa aplicação, inclusive seus testes. É por isso que já adicionamos a pasta script em nosso arquivo .gitignore.

Já podemos iniciar nosso servidor através do comando:

```
sh scripts/server.sh
```

Rodar nossos testes e2e:

```
sh scripts/test.sh
```

A inicialização do servidor e a execução de nossos testes e2e devem continuar a funcionar localmente, pois só mudamos a organização de nosso código, não o comportamento final da aplicação e dos testes.

12.6 Integração com o Travis

Nossa aplicação continua funcionando localmente depois das alterações realizadas na seção anterior. Agora precisamos completar nosso arquivo app/config/env/test.js com as informações que serão necessárias quando rodarmos nossa aplicação e nossos testes no Travis.

Se voltarmos para o arquivo config/env/test.js, veremos que ele depende de variáveis de ambientes que ainda não definimos. A ideia é definilas como variáveis de ambiente do Travis.

Logados no Travis, no topo direito da tela, há uma engrenagem que ao ser clicada exibirá uma lista com a opção settings. É através dessa opção que acessamos a página Environment Variables. Nela, podemos adicionar quantas variáveis de ambiente desejarmos, inclusive podemos optar para que seu conteúdo seja visível no log de construção do projeto (recomendado apenas para fins de depuração).

Vamos adicionar as seguintes variáveis:

- **SAUCE_USERNAME**: login do seu usuário (apenas o nome, sem @) no Sauce Labs. É utilizado pelo Protractor quando executarmos nosso testes e2e no Travis.
- **SAUCE_ACCESS_KEY**: sua chave no Sauce Labs. Acesse (https://saucelabs.com/account) e no canto inferior direito aparecerá seu Acess Key.
- **SELENIUM_USER**: nome de qualquer usuário do Github cuja senha você possui.
- **SELENIUM_USER_PASSWORD**: senha do usuário do Github.
- **CLIENT_ID**: cadastrado para a aplicação contatooh no GitHub.
- **CLIENT_SECRET**: gerado para a aplicação contatooh no GitHub.

Fig. 12.8: Variáveis de ambientes adicionadas no Travis

As variáveis de ambiente TRAVIS_JOB_NUMBER e TRAVIS_BUILD_NUMBER são disponibilizadas automaticamente pelo Travis e serão usadas pelo Sauce Labs através do Protractor para criar um identificador de nossos testes. Isso é interessante, porque saberemos que o teste X no Sauce Labs equivale ao build Y do Travis.

A variável NODE_ENV

Depois de atribuirmos todas as variáveis de ambiente necessárias no Travis, precisamos alterar nosso arquivo .travis.yml para que ele crie a variável NODE_ENV, só que desta vez ela terá como valor test:

```
env:
  global:
    - NODE_ENV: test
```

Comunicação com Sauce Labs

Também queremos que ele carregue o *proxy* que será usado para que nossos testes com Protractor sejam rodados no Sauce Labs:

```
addons:
  sauce_connect: true
```

Subindo a aplicação e executando nossos testes e2e

Estamos quase no □m! Precisamos ainda solicitar que, imediatamente após a execução de nossos testes com Karma, o Travis suba nossa aplicação e, em seguida, execute nossos testes e2e:

```
script:
 - karma start config/karma.config.js --single-run --browsers
 PhantomJS
 - node server &
 - protractor config/protractor.js
```

Subindo o MongoDB

Não podemos nos esquecer de subir uma instância do MongoDB para que possamos rodar nossos testes e2e:

```
services:
  - mongodb
```

Populando o banco antes dos testes

Precisamos popular o banco criado pelo Travis com alguns contatos antes de executarmos nossos testes. Para isso, criaremos um arquivo com essa □-nalidade, chamado contatooh/test/data.js. Desta vez, utilizaremos o driver do Mongo no lugar do Mongoose:

```
// contatooh/test/data.js

var MongoClient = require('mongodb').MongoClient;

var contatos = [
```

```
  {nome: "xyz1", email: 'xyz1@email.com.br' },
  {nome: "xyz2", email: 'xyz2@email.com.br' },
  {nome: "xyz3", email: 'xyz3@email.com.br' },
];

MongoClient.connect('mongodb://127.0.0.1:27017/contatooh_test',
  function(erro, db) {
    if(erro) throw err;

    db.dropDatabase(function(err) {
      if(err) return console.log(err);
      console.log('Banco apagado com sucesso')
      db.collection('contatos').insert(contatos,
      function(err, inserted) {
        if(err) return console.log(err);
        console.log('Banco populado com sucesso')
        process.exit(0);
      });
    });
  }
);
```

Por fim, precisamos pedir que o Travis execute nosso script antes de executar nossos testes e2e:

```
before_script:
  - node test/data.js
  - grunt
  - cd dist
```

.travis.yml: arquivo final

Caso o leitor tenha se perdido neste mundo de configurações, segue o .travis.yml final:

```
language: node_js
notifications:
  email: seuEmail
node_js:
```

```
  - 0.10
services:
  - mongodb
env:
  global:
    - NODE_ENV: test
addons:
  sauce_connect: true
before_install:
  - npm install -g bower@1.3
  - npm install -g grunt-cli@0.1
  - npm install -g karma-cli@0.0
  - npm install -g protractor@1.5
  - webdriver-manager update
  - bower install
before_script:
  - node test/data.js
  - grunt
  - cd dist
script:
 - karma start config/karma.config.js --single-run --browsers
  PhantomJS
 - node server &
 - protractor config/protractor.js
```

Adequando protractor.js ao novo ambiente

Nossa última alteração será em nosso arquivo contatooh/config/protractor.js para que ele utilize as variáveis de ambiente de□nidas em nosso servidor de integração. Não se preocupe quando você rodar o teste em ambiente de desenvolvimento, mesmo sem que o nosso arquivo config/env/development.js tenha de□nido algumas delas. Elas serão simplesmente ignoradas, algo justo, já que muitas delas só fazem sentido quando rodarmos nossos testes através do Sauce Labs, o que não faremos em ambiente de desenvolvimento.

```
// alterando contatooh/config/protractor.js

var config = require('./config')();
```

```
exports.config = {
  sauceUser : config.sauceUser,
  sauceKey : config.sauceKey,
  capabilities : {
    'name': config.sauceTestName,
    'browserName': 'chrome',
    'tunnel-identifier': config.travisJobNumber,
    'build': config.travisBuild
  },

  specs: ['../test/e2e/**/*Spec.js'],
  onPrepare: function() {
    browser.driver.get('http://localhost:3000');
    browser.driver.findElement(by.id('entrar')).click();
    browser.driver.findElement(by.id('login_field'))
      .sendKeys(config.seleniumUser);
    browser.driver.findElement(by.id('password'))
      .sendKeys(config.seleniumUserPassword);
    browser.driver.findElement(by.name('commit')).click();
  }
};
```

Agora já estamos pontos para executar nossos testes e2e pelo Travis. Logo após ter subido para o repositório do projeto todas as alterações que fizemos, verifique a construção do projeto pelo Travis. Se tudo foi configurado corretamente, nossos testes no Sauce Labs deve passar.

Podemos acessar nossa conta no Sauce Labs e verificar o status de nosso teste, screenshots e screencasts:

Fig. 12.9: Sauce Labs: testes realizados

CZ£ –¶ 13

Criando suas próprias diretivas

“Cuidado para não chamar de inteligentes apenas aqueles que pensam como você.”
– Ugo Ojetti

Nossa aplicação está pronta, excelente! Porém, realizaremos mais uma mudança. Precisamos envolver parte do conteúdo de nossas parciais contatos.html e contato.html dentro de um panel do Bootstrap. Este panel é construído com a combinação de uma série de elementos com classes especí□cas do Bootstrap:

```
<div class="panel panel-default">
  <div class="panel-heading">
    <h3 class="panel-title">Listagem de Contatos</h3>
  </div>
  <div class="panel-body">
```

```
    <!-- conteúdo do painel fica aqui -->
  </div>
</div>
```

Apesar de funcional, teremos este código repetido nas duas parciais e podemos correr o risco de esquecermos de adicionar uma ou outra classe, comprometendo o resultado nal.

O problema é que não temos uma maneira de reutilizar a marcação que vimos no lado do cliente, e precisaremos do título do painel variando de acordo com o contexto em que é utilizado. Imagine se o HTML suportasse a seguinte tag em seu vocabulário:

```
<!-- public/partials/contatos.html -->
<meu-painel titulo="Listagem de Contatos">
    <!-- conteúdo da parcial contatos.html -->
</meu-painel>
```

Com certeza poderíamos reutilizá-la em qualquer página, escondendo a complexidade de criação do painel do Bootstrap. Esta ideia não é novidade e muitos servidores web trabalham com o conceito de componente.

Um componente é um conjunto de código (Java, kC, HTML, CSS, JavaScript etc.) que encapsula determinado comportamento, podendo ser reutilizado facilmente em várias aplicações.

Vimos que o próprio AngularJS possui suas diretivas, que na verdade são componentes reutilizáveis que ampliam o vocabulário do navegador, ensinando-o novos truques. E se pudéssemos criar nossas próprias diretivas? É isso que veremos neste capítulo.

13.1 DDO D u h i Du Of u h

Criamos nossas próprias diretivas quando queremos um componente HTML ou um comportamento reutilizável, inclusive podemos querer ambos ao mesmo tempo. Também utilizamos este recurso do AngularJS quando precisamos interagir com o DOM. Daremos início nesta seção à criação de nossa primeira diretiva.

Vamos começar criando um módulo em separado que conterá nossas diretivas. Isso permitirá que elas sejam reutilizadas mais facilmente, bastando

importar o novo módulo, sem a necessidade de carregarmos todos os outros recursos da aplicação. Vamos criar o arquivo:

```
// public/js/directives/
// meus-componentes/meus-componentes.js

angular.module('meusComponentes', [])
.directive('meuPainel', function() {

  var directive = {};

  return directive;

});
```

Criamos o módulo meusComponentes, que não possui dependência alguma. Em seguida, chamamos a função directive, que recebe como primeiro parâmetro o nome da diretiva. Repare que seu nome utiliza o padrão *CamelCase*, porém, quando formos utilizá-la em nossa página, separaremos cada palavra por um hífen:

```
<meu-painel></meu-painel>
```

O segundo parâmetro da diretiva é uma função que sempre deve retornar um *directive definition object* (DDO). Nossa próxima tarefa será configurar este objeto.

A propriedade restrict

Nossa primeira configuração diz respeito à forma de utilização da diretiva. Queremos utilizá-la como (E)lemento ou como (A)tributo. Para isso, adicionamos em nosso DDO a propriedade restrict:

```
// public/js/directives/
// meus-componentes/meus-componentes.js

angular.module('meusComponentes', [])
.directive( 'meuPainel', function() {
```

```
  var directive =    {};

    directive.restrict        = "EA";

  return   directive;
});
```

Vejamos os dois exemplos:

```
<!-- como (E)lemento -->
<meu-painel></meu-painel>

<!-- como (A)tributo -->
<div  meu-painel ></div>
```

Mas onde de niremos a marcação de nossa diretiva? É o que veremos a seguir.

A propriedade template

Sabemos que o AngularJS terá que processar nossa diretiva substituindo-a pelo panel do Bootstrap. É por isso que nosso DDO possui a propriedade template. É nesta propriedade que de nimos a marca ção do nosso panel, e utilizamos a Angular Expression {{titulo}} no local da impressão do titulo:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

angular.module('meusComponentes', [])
.directive(    'meuPainel',    function() {

  var  directive    = {}

  directive.restrict        = 'EA';

  directive.template      =
      '<div class="panel panel-default">'        +
      '   <div class="panel-heading">'      +
```

```
        '      <h3 class="panel-title">{{titulo}}</h3>'          +
        '    </div>'     +
        '    <div class="panel-body">'       +
        '    </div>'     +
        '</div>';

    return directive;
});
```

Do jeito que fizemos, nossa diretiva só conseguirá obter o título de algum controller que gerencie o elemento do DOM no qual ela está incluída. Isso não faz muito sentido, pois se tivéssemos dois painéis gerenciados por um mesmo controller, não conseguiríamos atribuir um título diferente para cada um deles. Como resolver? É o que veremos na próxima seção.

13.2 Isolando o escopo

Queremos que nossa diretiva meu-painel receba seu título através do atributo titulo, nossa interface de comunicação com a diretiva:

```
<meu-painel titulo="Listagem de Contatos" >
  <!-- conteúdo do painel -->
<meu-painel>
```

Mas para que cada meu-painel de nossa página tenha seu próprio título, precisamos que todos possuam um escopo isolado.

Escopo isolado

O **escopo isolado** de uma diretiva é independente do escopo em que ela existe, isto é, não herda do escopo pai no qual está inserida. Isso evita que a diretiva tente acessar algo que esteja fora dos seus limites, evitando que bagunce o escopo pai.

As propriedades de uma diretiva com escopo isolado só podem ser usadas dentro dela, sendo invisíveis para o mundo exterior. Todas essas características permitem a criação de um componente que pode ser usado independente do lugar onde o inserimos, algo fundamental para seu reúso, inclusive permitirá que cada meu-painel tenha seu próprio título.

Podemos criar um escopo isolado adicionando a propriedade **scope** (não confunda com $scope do controller!) em nosso DDO. Esta propriedade armazena um objeto cujas chaves são acessíveis dentro da diretiva. Normalmente, armazenamos nessas chaves os valores passados através dos atributos de nossas diretivas:

```
<!-- apenas exemplo, não entra na aplicação ainda -->

<meu-painel titulo="Listagem de Contatos">
</meu-painel>

// apenas exemplo, não entra na aplicação ainda

  directive.scope = {
    titulo: '@titulo'
  };
```

O código anterior busca o valor do atributo titulo da diretiva meu-painel em uso.

API oZ o□u □ □

Quando criamos uma diretiva, sua API nada mais é do que seus atributos. São eles que permitem que a diretiva se comunique com o mundo exterior. Se quisermos um escopo isolado, porém sem expor, uma API usamos:

```
diretive.scope = true;
```

Você deve ter reparado no @antes do valor do chave titulo. Ele é um dos modi□cadores (veremos outros em breve) utilizados para indicar como os valores são passados do mundo externo para dentro do escopo isolado da diretiva. Neste caso, @indica que o valor que receberemos será sempre uma string, uma cópia do valor presente no atributo da diretiva em uso na view. É como se fosse uma relação unidirecional, onde nossa diretiva pode ler este valor vindo do escopo pai, porém qualquer alteração nele dentro da diretiva não se propagará para o pai:

```
<!--
 Parâmetro válido, a string "Listagem de Contatos"
 é passada para a diretiva
-->
<meu-painel titulo="Listagem de Contatos">
</meu-painel>

<!--
 Parâmetro válido, o resultado da interpolação
 é passado para a diretiva
-->
<meu-painel titulo="Listagem de {{nome}}">
</meu-painel>

<!--
 Parâmetro válido, o resultado da AE
 é passado para a diretiva
-->
<meu-painel titulo="{{nome}}}">
</meu-painel>
```

Outro ponto a destacar é quando o atributo da diretiva na view é igual ao nome da propriedade em nosso escopo isolado, podemos omitir o nome do atributo deixando apenas @:

```
directive.scope = {
  titulo: '@'
};
```

Nossa diretiva ficará assim:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

angular.module('meusComponentes', [])
.directive('meuPainel', function() {

  var directive = {}

  directive.restrict = 'EA';
```

```
  directive.scope = {
    titulo: '@'
  };

  directive.template =
      '<div class="panel panel-default">' +
      '  <div class="panel-heading">' +
      '    <h3 class="panel-title">{{titulo}}</h3>' +
      '  </div>' +
      '  <div class="panel-body">' +
      '  </div>' +
      '</div>';

  return directive;
});
```

Excelente! Já podemos testar nossa diretiva. Para isso, precisamos importá-la em nossa página app/views/index.ejs como último script, mas ainda dentro do comentário especial do grunt-usemin:

```
<!-- app/views/index.ejs -->

<!-- build:js js/index.min.js -->
<!-- scripts anteriores omitidos -->
<script src="js/directives/meus-componentes/
  meus-componentes.js">
</script>
<!-- endbuild -->
```

Como criamos um novo módulo, precisamos importá-lo no módulo principal da aplicação:

```
// public/js/main.js

angular.module('contatooh',
  ['ngRoute', 'ngResource', 'meusComponentes'])
  .config(function($routeProvider) {
    // código omitido
  });
```

Por fim, vamos envolver apenas a `div` da lista de contatos. Utilizaremos a diretiva como elemento:

```
<!-- public/partials/contatos.html -->


<!-- tags anteriores omitidas -->

<p ng-show="contatos.length">
  Contatos cadastrados: {{contatos.length}}
</p>

<meu-painel titulo="Listagem de contatos">

  <!-- apenas a div da tabela dentro da diretiva -->
  <div class="table-responsive">

    <table class= "table table-hover">
      <!-- dados da tabela omitido -->
    </table>

  </div>
<meu-painel>
```

O [illegible] HTML

Muitas vezes, é exigido que sua página passe por validadores que verifiquem a estrutura do seu documento, mais notadamente os da W3C. Com certeza, as diretivas do AngularJS não passarão no teste. Para contornar este problema, todas as diretivas do AngularJS podem ser prefixadas com data-. Este prefixo indica para o validador que estamos utilizando um custom attribute, algo aceito dentro do HTML5:

```
<p data-ng-repeat="objeto in objetos">
```

No exemplo do nosso painel, poderíamos utilizar a diretiva no seu formato atributo, inclusive prefixando-a com data-:

```
<div data-meu-painel>
    <!-- conteúdo aqui dentro -->
</div>
```

Visualizando o resultado de nossa diretiva, temos:

Fig. 13.1: Diretiva em ação, porém nenhum conteúdo é exibido

Repare que a diretiva é renderizada, porém nada é exibido! É como se ela ignorasse tudo o que colocamos dentro dela. A solução deste problema encontra-se na próxima seção, curioso?

13.3 [illegible]

Na seção anterior, criamos nossa diretiva meu-painel, porém seu conteúdo não é exibido. Isso ocorre porque, ao avaliar a diretiva, o AngularJS troca

aquele fragmento do DOM por um novo, ignorando tudo o que tinha antes lá. Porém, podemos solicitar que ele mantenha o conteúdo do elemento adicionando em nosso DDO a propriedade transclude:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

angular.module('meusComponentes', [])
.directive( 'meuPainel', function() {

  var directive = {}

  directive.restrict = 'EA';

  directive.scope = {
    titulo: '@'
  };

  directive.transclude = true;

  directive.template =
      '<div class="panel panel-default">'
      + '<div class="panel-heading">'
        + '<h3 class="panel-title">{{titulo}}</h3>'
      + '</div>'
      + '<div ng-transclude class="panel-body">'
      + '</div>'
      + '</div>';

  return directive;
});
```

Repare que, além de adicionarmos a propriedade directive.transclude=true, também adicionamos a diretiva ng-transclude no elemento de nosso template cujo conteúdo deverá ser preservado.

Testando mais uma vez, o painel é exibido, maravilha!

Fig. 13.2: Diretiva em ação

Karma: atualizando arquivo de configuração

Não podemos nos esquecer de incluir a pasta

../public/js/directives/**/*.js

no arquivo de configuração do Karma, caso contrário, nossos testes de unidade falharão por não encontrarem o módulo meusComponentes, já que o importamos no módulo principal da aplicação:

```
// config/karma.config.js
    files: [
    '../public/vendor/angular/angular.js',
    '../public/vendor/angular-mocks/angular-mocks.js',
    '../public/vendor/angular-resource/angular-resource.js',
    '../public/vendor/angular-route/angular-route.js',
    '../public/js/main.js',
    '../public/js/controllers/**/*.js',
    '../public/js/services/**/*.js',
    '../public/js/directives/**/*.js',
    '../test/spec/**/*Spec.js'
    ],
```

Podemos melhorar ainda mais o design de nossa diretiva, assunto da próxima seção.

13.4 Su Z Z o DDO ou u u Z u

Você deve ter reparado que escrever o template do painel dentro da diretiva torna sua manutenção mais trabalhosa. Imagine se o template fosse ainda

mais complexo? Aliás, temos misturadas em um único arquivo a configuração da diretiva e sua apresentação. Podemos isolar o template da diretiva em seu próprio arquivo HTML, com a diferença de que não usaremos mais a propriedade template, mas templateUrl que aponta para o arquivo.

Primeiro, vamos mover a marcação do nosso template para dentro do novo arquivo public/js/directives/meus-componentes/meu-painel.html:

```
<!-- public/js/directives/
  meus-componentes/meu-painel.html -->

<div class="panel panel-default">
  <div class="panel-heading">
    <h3 class="panel-title">
      {{titulo}}
    </h3>
  </div>
  <div ng-transclude class="panel-body">
  </div>
</div>
```

Agora, em nossa diretiva, vamos substituir a propriedade template por templateUrl apontando para nosso novo arquivo:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

angular.module('meusComponentes', [])
.directive('meuPainel', function() {

  var directive = {}

  directive.restrict = 'EA';

  directive.scope = {
    titulo: '@'
  };

  directive.transclude = true;
```

```
    directive.templateUrl =
        'js/directives/meus-componentes/meu-painel.html';

    return directive;
});
```

Nossa diretiva deve continuar com o mesmo resultado, pois mudamos apenas a maneira com que organizamos seu código.

Agora podemos usar a nossa diretiva em qualquer página, inclusive em nossa parcial public/partials/contato.html. Porém, antes de qualquer reúso, vamos criar mais uma diretiva, desta vez o botão remover de nossa lista.

13.5 MZ o u

Em nossa parcial public/partials/contatos.html, temos o seguinte botão responsável pela remoção de um contato:

```
<button ng-click="remove(contato)"
    class="btn btn-warning">
    Remover
</buttom>
```

Queremos que ele se torne esta diretiva:

```
<meu-botao-aviso nome="Remover"
    acao="remove(contato)">
</meu-botao-aviso>
```

Vamos adicionar uma nova diretiva em nosso módulo meusComponentes, que se chamará meuBotaoAviso. Criaremos um escopo isolado com as propriedades nome e acao:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

angular.module('meusComponentes', [])
```

```
.directive('meuPainel', function() {
  // código omitido
})
.directive('meuBotaoAviso', function() {

    var directive = {}

    directive.restrict = 'E';

    directive.scope = {
      nome: '@',
      acao : '&'
    };

    return directive;
});
```

Diferente de meu-painel, queremos que a nova diretiva seja utilizada apenas como elemento em nossas páginas. Além disso, utilizamos o modificador & para a propriedade acao. O modificador & nos permite avaliar ou invocar uma expressão (função) no escopo do elemento no qual a diretiva está inserida. Faz todo sentido, porque, quando o usuário clicar no botão remover, queremos que o código executado seja aquele no escopo do controller. A diretiva não fará ideia do resultado da expressão.

Agora só nos resta definir o template do nosso botão. Desta vez, manteremos sua declaração dentro da própria definição da diretiva com a propriedade template:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

angular.module('meusComponentes', [])
.directive('meuPainel', function() {
  // código omitido
})
.directive('meuBotaoAviso', function() {

    var directive = {}
```

```
    directive.restrict = 'E';

    directive.scope = {
      nome: '@',
      acao : '&'
    };

    directive.template =
      '<button ng-click="acao()" class="btn btn-warning">'
      + '{{nome}}'
    + '</buttom>';

    return directive;
});
```

Podemos substituir o componente do Bootstrap pelo nosso botão:

```
<!-- public/partials/contatos.html -->

<meu-botao-aviso nome="Remover"
  acao="remove(contato)">
</meu-botao-aviso>
```

Toda vez que precisarmos de um botão de aviso, poderemos recorrer à nossa diretiva. Que tal tentarmos algo mais so□sticado na próxima seção?

13.6 MZ□□□Z□o□ DOM Z□ □Z□□ ou o□u □ □

Nossa aplicação está □cando cada vez melhor, porém experimente alterar um contato. Com certeza, ele será alterado, nosso formulário limpo, porém o foco do cursor □cará sob o botão salvar. Queremos que ele □que sob o botão voltar, um requisito de UX válido, não? Sabemos que isso pode ser feito via JavaScript, acessando o elemento do DOM:

```
// ContatoController
// código ilustrativo, não entra na aplicação

$scope.salva = function() {
```

```
        $scope.contato.$save()
        .then(function() {
          // código anterior omitido

          // manipulando DOM no controller, please, no!

          document.querySelector('#botao-voltar')
            .focus();
        })
};
```

Porém, a manipulação de DOM dentro de nossos controllers os tornará difíceis de testar, inclusive, quase tudo do AngularJS que vimos até agora foi para evitarmos este tipo de manipulação para focarmos apenas na lógica da aplicação. É aí que a diretiva do Angular vem nos salvar.

Na diretiva podemos realizar manipulação de DOM à vontade quando necessário. Por ser um pedaço de código isolado e totalmente desacoplado de nossos controllers, não há impacto na testabilidade desses últimos.

Vamos criar uma diretiva que será capaz de atribuir o foco em qualquer elemento da página contanto que o valor do atributo focus seja true. Por exemplo:

```
<a id="botao-voltar" meu-focus focus="btnBackFocus"
  href="#/" class="btn btn-default">
    Voltar
</a>
```

Nesse exemplo, temos a diretiva meu-focus

```
   apenas ilustrativo, ainda não entra
   no código da aplicação.
*/

$scope.salva = function() {

    $scope.contato.$save()
    .then(function() {
      // código anterior omitido

      // sem manipulação de DOM, yeh!
      $scope.btnBackFocus = true;
    })
};
```

Agora que já temos uma ideia da API de nossa diretiva, já podemos criá-la:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

angular.module('meusComponentes', [])
.directive('meuPainel', function() {
  // código omitido
})
.directive('meuBotaoAviso', function() {

  // código omitido
})
.directive('meuFocus', function() {

    var directive = {};

    directive.restrict = 'A';

    directive.scope = {
      focus: '='
    };
```

```
    return directive;
});
```

Consegue ver a novidade? Usamos focus: '=' no escopo isolado da diretiva. Este modificador permite que nossa diretiva obtenha do atributo focus o valor da expressão atribuída a ele, e qualquer mudança realizada pela diretiva neste valor atualizará também o valor no escopo pai. Este modificador é uma espécie de two-way data binding, no qual tanto o escopo pai quanto o escopo isolado da diretiva têm a mesma referência para a expressão. Porém, com este modificador, não podemos usar {{}} como valor da diretiva, apenas expressões:

```
<!-
  Parâmetro válido: a diretiva recebe a expressão
  "btnBackFocus", avaliada N vezes dentro da diretiva.
-->
<a meu-focus focus="btnBackFocus" Voltar</a>

<!-
  Parâmetro inválido: a expressão é avaliada, retornando true
  ou false. Em seguida, o retorno é passado para a diretiva.
  Neste caso, a diretiva não tem mais uma referência
  para a expressão.
-->
<a meu-focus focus="{{btnBackFocus}}" Voltar</a>
```

E [illegible]

Se tivéssemos utilizado @, o valor de focus seria sempre a string "btnBackFocus", e não o valor corrente da expressão.

E agora? Como teremos acesso ao elemento do DOM para podermos focá-lo? Primeiro, precisamos de uma introdução sobre a maneira pela qual o AngularJS processa suas diretivas quando são encontradas.

Um pouco sobre diretiva e suas fases

O AngularJS processa diretivas em duas fases. A primeira é a **compile phase**. Nesta fase, o AngularJS transforma o template de nossas diretivas em elementos do DOM para, logo em seguida, adicioná-los no documento, substituindo ou aprimorando elementos já existentes. Raramente interagimos com esta fase, pois ela foi criada para resolver problemas de performance quando utilizamos o mesmo template de origem (antes de compilar) várias vezes, como é o caso da diretiva ng-repeat. Toda compilaç ão retorna uma função de link, o que caracteriza o início da segunda fase.

A segunda fase é chamada de **link phase**. É nesta fase que o escopo da diretiva é criado, observações de mudanças de propriedades podem ser feitas, inclusive podemos ter acesso ao escopo criado e ao elemento do DOM ao qual a diretiva foi associada. É justamente o que precisamos para podermos atribuir o foco ao elemento:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

// código anterior omitido
.directive( 'meuFocus', function() {

    var directive = {};

    directive.restrict = 'A';

    directive.scope = {
      focus: '='
    };

    directive.link = function(scope, element) {
    });

    return directive;
});
```

A [illegible]

Os argumentos da função link são posicionais, não injeções. Se tivéssemos invertido os parâmetros no exemplo anterior, nosso código deixaria de funcionar.

No exemplo anterior, estamos interessados apenas no escopo da diretiva e no elemento do DOM ao qual ela está associada. Precisamos solicitar que o Angular monitore o valor da propriedade scope.focus.

13.7 A [illegible]

Na seção anterior, temos o esqueleto do código de nossa diretiva com a funcionalidade de focar elementos de nossa página, inclusive adicionamos na API de nossa diretiva a propriedade focus. Porém, toda vez que a propriedade focus mudar, precisaremos testar se o seu valor é true para depois focarmos o elemento do DOM.

Realizamos essa observação através da função scope.$watch, que recebe como parâmetro o nome da propriedade que desejamos observar e uma função de callback que será executada a cada mudança:

```
// public/js/directives/meus-componentes/
// meus-componentes.js

// código anterior omitido
.directive('meuFocus', function() {
    var directive = {};

    directive.restrict = 'A';

    directive.scope = {
      focus: '=focus'
    };

    directive.link = function(scope, element) {
      scope.$watch('focus', function() {
```

```
            if (scope.focus) {
                element[0].focus();
                scope.focus = false;
            }
        });
    };
    return directive;
});
```

Dentro da função de callback, realizamos um teste para saber o valor atual de scope.focus. Em seguida, chamamos a fun ção element[0].focus() para focarmos o elemento.

> Na função passada como segundo parâmetro, temos acesso ao valor atual e antigo do elemento monitorado:
>
> ```
> scope.$watch('focus', function(valorAtual, valorAntigo) {
> });
> ```

Acessamos o elemento como se ele fosse um array, porque internamente o Angular utiliza o jQLite (https://code.google.com/p/jqlite/) , um subconjunto da API Sizzle (http://sizzlejs.com/) utilizada pelo jQuery para nos ajudar com a manipulação de DOM. Porém, o jQLite não possui a função focus(). O que □zemos foi acessar o elemento do DOM encapsulado pelo jQLite.

> **u □u □u □□ h□□□ □Q □u □□ Of □u h□**
>
> Se tivéssemos carregado o jQuery em nossa página antes do script do AngularJS, element se tornaria um jQuery Object, permitindo acesso a todas as funções do jQuery, incluindo a função focus.

Já podemos alterar nossa parcial public/contato.html substituindo a marcação do botão voltar

public/js/controllers/ContatoController.js a propriedade btnBackFocus:

```
<!-- public/partials/contato.html -->

<a id="botao-voltar" meu-focus focus="btnBackFocus"
  href="#/" class="btn btn-default">
    Voltar
</a>

// public/js/controllers/ContatoController.js

    $scope.contato.$save()
    .then(function() {
      // código omitido

      // novidade entra aqui, última linha da função!
      $scope.btnBackFocus = true;
    })
```

Excelente! Agora, basta cadastramos um novo contato e vermos que o botão voltar recebe o foco logo em seguida.

Toda vez que usamos $watch. gastamos poder computacional, sendo assim, devemos ser parcimoniosos com seu uso. O ideal é utilizá-lo quando não houver outra alternativa. Por exemplo, para resolvermos o problema do foco que acabamos de ver, podemos trabalhar com o event bus do AngularJS, assunto da próxima seção.

13.8 T□Z f Z□□Z□o□ h□□ gf□□Z ohZ□□

Vamos modi□car a diretiva que □zemos na seção anterior. No lugar de usarmos o tão caro $watch, utilizamos o event bus do AngularJS.

Primeiro, em nosso controller

public/js/controllers/ContatoController.js

vamos substituir a linha

```
$scope.btnBackFocus = true;
```

por este novo código:

```
$scope.$broadcast('contatoSalvo');
```

É através da função $broadcast de nosso escopo que disparamos um evento. A função recebe como parâmetro o nome do evento que desejamos disparar, em nosso caso, contatoSalvo. Vamos alterar também a API de nossa diretiva que receberá como parâmetro a propriedade evento.

Vamos ajustar seu uso na página public/partials/contato.html:

```
<a id="botao-voltar" meu-focus evento="contatoSalvo"
  href="#/" class="btn btn-default">
    Voltar
</a>
```

Repare que a nova API de nossa diretiva recebe como parâmetro um evento. O que precisamos fazer agora é alterar nossa diretiva para que seja capaz de responder ao evento que será disparado em nosso controller:

```
// public/js/directives
  /meus-componentes/meus-componentes.js

// código anterior omitido
.directive( 'meuFocus', function() {

    var directive = {};
    directive.restrict = 'A';
    directive.scope = {
      evento: '@'
    };

    directive.link = function(scope, element) {
      scope.$on(scope.evento, function() {
        element[0].focus();
      });
    };
    return directive;
});
```

O escopo privado da diretiva receberá apenas a string com o nome do evento que desejamos escutar, por isso, usamos o modi□cador @. Em seguida, dentro da nossa função link, escutamos ao evento passado como parâmetro através da função $on.

gf□□Z ohZ□□ □ gu □□□

Além da função $broadcast, existe também a fun ção $emit. A diferença é que o evento disparado pelo primeiro desce na hierarquia do escopo, afetando apenas seus elementos □lhos. Já o segundo sobe na hierarquia.

O primeiro parâmetro da função $on é o nome do evento que desejamos escutar, o segundo, uma função de callback executada toda vez que o evento é disparado. Em nosso caso, quando o evento for disparado, o elemento ao qual a diretiva foi associada receberá o foco. Pronto, agora é só testarmos o resultado. Usamos uma solução mais barata e elegante para resolver o problema do foco do botão.

Excelente, temos nossas diretivas prontas, mas será que nos esquecemos de algo? Aprendemos a testar nossos controllers na seção 11.2, inclusive utilizamos o Karma para automatizar este processo. Na próxima seção, aprenderemos a testar nossas diretivas.

13.9 Tu □□ Z□o□ o□□u □ □□□

Aprendemos a criar diretivas na seção anterior, e realizamos manipulação de DOM através delas. Porém, todo código que interage com a API DOM tem certa fragilidade. É por isso que é extremamente importante a realização de testes para termos a garantia de que nosso código será compatível com os mais diversos navegadores do mercado. Nesta seção, criaremos um teste para cada diretiva que criamos.

Vamos começar adicionando o script de nossa diretiva ao arquivo de con□guração config/karma.config.js:

```
files: [
// código anterior omitido
'../public/js/directives/**/*.js',
'../test/spec/**/*Spec.js',
],
```

Agora, já podemos criar nosso arquivo de teste test/spec/meusComponentesSpec.js. Em seguida, utilizaremos a já conhecida função describe para descrever nosso teste e a função beforeEach para prepararmos o ambiente antes de rodarmos nossas Specs:

```
describe('meuBotaoAviso', function () {

  var $scope;
  var element;

  beforeEach( function() {

    module('meusComponentes');

    inject(function($rootScope, $compile) {

      $scope = $rootScope.$new();
    });
  });
});
```

Carregamos o módulo meusComponentes e solicitamos ao serviço de injeção de dependência do AngularJs os artefatos $rootScope e $compile. O primeiro não é novidade para nós, porém o segundo é o serviço responsável pela compilação das diretivas. Nosso próximo passo será criar um elemento do DOM a partir de uma string que re□ete nossa diretiva em uso. Fazemos isso através da função angular.element:

```
describe('meuBotaoAviso', function () {

  var $scope;
  var element;
```

```
  beforeEach(function() {

    module('meusComponentes');

    inject(function($rootScope, $compile) {

      $scope = $rootScope.$new();
      element = angular.element(
        '<meu-botao-aviso nome="Remover" acao="remove()">'
      );

    });
  });
});
```

O resultado de angular.element será um elemento do DOM que não faz parte de nenhum documento. Esta etapa é importante porque ela lançará um erro caso a marcação que a string armazena seja inválida.

Agora, precisamos compilar este elemento. Este processo consiste em passá-lo como parâmetro para o serviço $compile. A função retornará uma função de link. Lembre-se que é na link phase que associamos um escopo ao elemento, sendo assim, passamos nosso $scope que criamos a partir do $rootScope. Vejamos um exemplo:

```
var lnkFunc = $compile(element);
lnkFunc($scope);
```

Podemos realizar esse processo em uma tacada só, evitando a criação da variável lnkFunc:

```
describe('meuBotaoAviso', function () {

  var $scope;
  var element;

  beforeEach(function() {

    module('meusComponentes');
```

```
    inject(function($rootScope, $compile) {

      $scope = $rootScope.$new();
      element = angular.element(
        '<meu-botao-aviso nome="Remover" acao="remove()">');

      $compile(element)($scope);
      $scope.$digest();

    });
  });
});
```

Veja que imediatamente após a compilação da diretiva chamamos $scope.$digest. Lembre-se que o AngularJS suporta data binding, permitindo que nossa view apresente o valor atualizado do modelo de um escopo. Quem monitora o modelo veri□cando por mudanças são os watchers, e já aprendemos a utilizá-los em uma de nossas diretivas personalizadas. Todos os watchers da aplicação são avaliados no ciclo digest chamado automaticamente pelo AngularJS. Porém, quando testamos nossas diretivas, somos os responsáveis por sua invocação.

Pronto, a variável element guarda nossa diretiva compilada e processada pelo AngularJS. Já podemos criar nossa primeira Spec para testar nossa diretiva:

```
describe('meuBotaoAviso', function () {

  var $scope;
  var element;

  beforeEach( function() {
    // código omitido
  });

  it('deve criar um botão de aviso com texto e função',

    function() {
```

```
        expect(element.text()).toContain('Remover');
        expect(element.attr('acao')).toBe('remove()');
      }
    );

});
```

Usamos as funções element.text e element.attr porque element é um objeto criado pelo jQlite que nos fornece uma API cômoda para realizarmos manipulação de DOM.

Dentro da pasta contatooh, vamos rodar o Karma no terminal com o comando:

```
karma start config/karma.config.js
```

Vamos deixá-lo aberto enquanto criamos novos testes. O teste que acabamos de criar passará.

Agora, vamos criar um novo teste para nossa diretiva meu-foco. Para isso, adicionaremos dentro do mesmo arquivo

```
public/js/directives/meus-componentes
  /meus-componentes.js
```

outra chamada à função describe:

```
describe('meuBotaoAviso', function () {
  // código omitido
});

describe('meuFocus', function() {

  var $scope;
  var element;
  var evento = 'contatoSalvo';

  beforeEach(function() {

    module('meusComponentes');
    inject(function($rootScope, $compile) {
```

```
        $scope = $rootScope.$new();

        element = angular.element(
          '<button meu-focus evento="' + evento +
          '">Voltar</button>');

        $compile(element)($scope);
        $scope.$digest();
      });
    });

});
```

Repare que esse código é muito parecido com o código do teste anterior, porém, precisamos resolver o seguinte problema. Para testarmos se nosso elemento está focado, precisamos adicioná-lo ao documento, caso contrário, não poderemos recorrer à chamada document.activeElement para descobrir qual é o elemento focado na página. Fazemos isso promovendo document para um objeto do jQlite para que possamos utilizar a função append:

```
// código anterior omitido
describe('meuFocus', function() {

  // código anterior omitido

  it('Deve focar o botão', function() {

    angular.element(document.body).append(element);

  });

});
```

Ótimo! Nosso elemento compilado agora faz parte do documento. Agora, precisamos disparar o evento contatoSalvo e verificar se o texto de document.activeElement corresponde ao texto do nosso botão:

```
// código anterior omitido
describe('meuFocus', function() {

  // código anterior omitido

  it('Deve focar o botão',       function() {

    angular.element(document.body).append(element);

    $scope.$broadcast(evento);

    expect(angular.element(document.activeElement)
      .text()).toBe('Voltar');

  });

});
```

Mais um teste pronto! Se você não fechou o Karma, receberemos um feedback em tempo real sobre status do nosso teste.

> Não é incomum termos vários Spec para o alvo de nosso teste. Por exemplo, poderíamos testar se a nossa diretiva meu-focus não ganha o foco caso o evento contatoSalvo não tenha sido disparado. A ideia é testar os possíveis casos cujos resultados sabemos de antemão.

Por □m, só nos resta testar nossa diretiva meu-painel. Teremos uma seção dedicada para este teste.

13.10 L□oZ□o□ h□□□ u □□□Z□ u U □□ u □□ u □□ u □

Nesta seção, vamos elaborar o teste da diretiva meu-painel adicionando mais uma chamada à função describe em nosso arquivo test/spec/meusComponentesSpec.js:

```
describe('meuBotaoAviso', function       () {
  // código omitido
```

```
});

describe('meuFocus', function() {
  // código omitido
});

describe('meuPainel', function() {

  var $scope;
  var element;

  beforeEach(function() {

    module('meusComponentes');
    inject(function($compile, $rootScope) {

      $scope = $rootScope.$new();

      element = angular.element(
        '<meu-painel titulo="Principal"><p>Oi</p></meu-painel>'
      );

      $compile(element)($scope);
      $scope.$digest();
    });
  });

});
```

A ideia é simples: verificamos se o conteúdo das tags `h3` e `p` correspondem ao título e ao conteúdo do painel, respectivamente. Porém, recebemos a seguinte mensagem de erro no terminal quando nosso teste é executado:

Error: Unexpected request: GET js/directives/meus-componentes/meu-painel.html No more request expected

Este erro acontece porque utilizamos `templateUrl` em nossa diretiva. O AngularJS tenta realizar uma requisição Ajax para baixar o template `meu-painel.html` e, como não temos o servidor rodando, o erro acontece.

Vimos que é vantajoso utilizarmos templateURL e não iremos abdicar dele! Para solucionarmos este problema, precisamos fazer com que o meu-painel.html esteja presente no cache de templates do AngularJS antes de o nosso código rodar. Este cache é usado por debaixo dos panos toda vez que o template de uma diretiva é buscado, justamente para evitar novas requisições.

No lugar de interagirmos com o serviço $templateCache que o AngularJs disponibiliza, utilizaremos o Karma para pré-processar todos os nossos templates, convertendo-os para um arquivo .js e colocando-os automaticamente dentro do cache de templates do AngularJS.

Primeiro, precisamos instalar o plugin do Karma responsável por este pré-processamento:

```
npm install karma-ng-html2js-preprocessor@0.1 --save-dev
```

Em seguida, precisamos adicionar no karma.config.js o caminho '../public/js/directives/meus-componentes/*.html', que contém nossos templates:

```
files: [
'../public/vendor/angular/angular.js',
'../public/vendor/angular-mocks/angular-mocks.js',
'../public/vendor/angular-resource/angular-resource.js',
'../public/vendor/angular-route/angular-route.js',
'../public/js/main.js',
'../public/js/controllers/**/*.js',
'../public/js/services/**/*.js',
'../public/js/directives/**/*.js',
'../test/spec/**/*Spec.js',
'../public/js/directives/meus-componentes/*.html'
],
```

Precisamos ainda indicar que nossos templates devem ser pré-processados pelo nosso plugin:

```
preprocessors : {
  '../public/js/directives/**/*.html': 'ng-html2js'
},
```

Isso ainda não é suficiente. Ainda no karma.config.js, precisamos adicionar o plugin na lista de plugins utilizados:

```
plugins : [
  'karma-ng-html2js-preprocessor',
  'karma-chrome-launcher',
  'karma-phantomjs-launcher',
  'karma-jasmine'
],
```

Estamos quase lá! Precisamos resolver o seguinte problema: quando o Karma pré-processar nosso arquivo meu-painel.html, ele guardará no cache de templates o arquivo com seu caminho absoluto, como neste exemplo:

```
/Users/flaviohenriquealmeida/Desktop/trab/reposi
torios/contatooh/public/js/directives/meus-
componentes/meu-painel.html.js
```

O problema é que o templateUrl de nossa diretiva aponta para js/directives/meus-componentes/meu-painel.html.
Essa discrepância fará com que nosso template não seja localizado. Para resolvermos isso, utilizaremos a propriedade ngHtml2JsPreprocessor para solicitarmos a remoção de todo o caminho /Users/flaviohenriquealmeida/Desktop/trab/repositorios/contatooh/ através de uma expressão regular, inclusive daremos um nome para o módulo que ele gerará no final para que possamos importá-lo em nossos testes:

```
ngHtml2JsPreprocessor: {
  moduleName: 'templates',
  stripPrefix: '.*/public/'
},
```

Pronto, terminamos de ajustar o karma.config.js. Agora, precisamos importar o módulo templates dentro da função beforeEach do teste do painel:

```
// código omitido
describe('meuPainel', function() {
```

```
  var $scope;
  var element;

  beforeEach(function() {

    module('meusComponentes');
    // importando o módulo gerando pelo Karma
    module('templates');
    // código omitido
  });
  // código omitido
});
```

Precisamos reiniciar o Karma, que logo em seguida rodará nosso último teste. Excelente!

Diretiva personalizada é um assunto muito amplo e muito badalado em comunidades. O leitor pode se aprofundar ainda mais no assunto consultando a própria documentação do Angular em https://docs.angularjs.org/guide/directive.

No próximo capítulo veremos a última etapa do workflow de nossa aplicação: o deploy.

CZ£ –¶ 14

Deploy da aplicação

"Just ship, baby"
– Kent Back

Nossa aplicação está pronta e devidamente testada, mas onde a hospedaremos? Em outras palavras: onde faremos o seu *deploy*? Além disso, será que precisaremos con□gurar toda a infraestrutura necessária para que nossa aplicação rode, por exemplo, instalar e con□gurar o MongoDB e o Node.js?

Aprendemos a utilizar serviços na web como nosso servidor de integração contínua Travis e o Sauce Labs (TaaS), retirando de nós, desenvolvedores, a responsabilidade de mantê-los. Será que não podemos fazer a mesma coisa com o deploy da aplicação? Sim, podemos!

Neste capítulo, aprendemos a realizar o deploy de nossa aplicação, mas antes precisamos de□nir com que frequência ele será realizado. Este é o as-

sunto da primeira seção deste capítulo.

14.1 Du [illegible]

Não é raro o desenvolvedor realizar o deploy de sua aplicação manualmente, por exemplo, parando a aplicação, abrindo uma conexão FTP, subindo seus arquivos, alterando con□gurações do ambiente de produção e iniciando a aplicação novamente.

Aprendemos ao longo deste livro que tudo que é feito manualmente pelo ser humano está sujeito a erros. Por mais que exista alguém perfeito que nunca erre, o fator tempo será sempre uma constante. Quanto tempo será gasto preparando todo este processo? Quanto tempo o cliente esperará até receber um novo *release* da aplicação?

Hoje, a cada *commit*, nosso servidor de integração dispara o *build* da nossa aplicação e executa uma série de testes para garantir a qualidade de nosso so□ware. Que tal se após o sucesso de nossos testes realizássemos o deploy da aplicação? **Um commit, um deploy!** Pode ser que isso não faça sentido dentro do contexto de outras aplicações, mas a aplicação Contatooh tem uma estrutura que permite isso. Sendo assim, na próxima seção conheceremos o serviço escolhido para realizarmos o deploy da aplicação.

14.2 O□u □S□□□□ : □Z□ □□□□Z□ Z □u □hu (PZZS)

OpenShi□ (http://www.openshi□.com/) é uma *platform as a service (PaaS)* criado pela Red Hat. Uma PaaS facilita o deploy da aplicação reduzindo drasticamente a complexidade envolvida no processo, inclusive fornece ferramentas que lidam com os detalhes de infraestrutura sem que nos preocupemos com eles.

Criando sua conta

Precisamos, antes de mais nada, criar nossa conta em http://www.openshi□.com. Você receberá um e-mail com a con□rmação de cadastro. Depois, basta se logar, que uma página de boas-vindas será exibida:

Usaremos o próprio servidor do OpenShi□, por isso podemos teclar ENTERpara esta pergunta.

```
2) You can add more servers later using 'rhc server'.
Login to openshift.redhat.com: suaConta
Password: *******
```

O rhc precisa do login e da senha de sua conta no OpenShi□ para que ele possa interagir com sua aplicação.

```
3) OpenShift can create and store a token on disk which allows
to you to access the server without using your password. The key
is stored in your home directory and should be kept secret. You
can delete the key at any time by running 'rhc logout'.
Generate a token now? (yes|no) yes
```

O rhc se encarregará de criar uma chave para que ele possa se comunicar seguramente com nossa aplicação no OpenShi□, por isso responderemos com yes. Pronto, agora podemos criar nossa aplicaç ão na próxima seção através do rhc.

14.3 C□□□□□□Z□o□ □Z□f□u □□ u ou ou □□□□

Podemos criar uma aplicação pela interface web, porém a maneira mais rápida e recomendada é através do rhc. Com o comando rhc create-app, criamos nossa aplicação. Ele recebe como primeiro parâmetro o nome da aplicação e, como segundo, a plataforma utilizada. Por □m, podemos adicionar um ou mais cartridges que correspondem a recursos de infraestrutura como o MongoDB.

```
rhc create-app contatooh nodejs-0.10 mongodb-2.4
```

O rhc criará um repositório local para nossa aplicação que aponta para o repositório remoto criado no OpenShi□ (não é criado no GitHub!). OpenShi□ utiliza o git para que possamos subir as alterações de nossa aplicação. Porém, veremos mais adiante que não trabalharemos com esse repositório localmente. A ideia é que o Travis realize automaticamente o deploy de nosso projeto caso ele passe em todos os testes.

Excelente! Se quisermos visualizar o log da aplicação sempre atualizado usamos o comando:

```
rhc tail contatooh
```

Esse recurso é interessante, principalmente quando queremos descobrir problemas no deploy da aplicação.

14.4 Preparando a aplicação para deploy

Tudo o que fizemos até agora é suficiente para o deploy? Com certeza, não. Ainda falta saber qual o endereço do nosso banco de dados, o domínio da aplicação, seu IP e porta. Não podemos simplesmente arbitrar esses valores. É por isso que o OpenShift disponibiliza essas informações também através de variáveis de ambiente, são elas:

- **OPENSHIFT_MONGODB_DB_URL**: endereço do MongoDB criado pelo OpenShift;
- **OPENSHIFT_NODEJS_PORT**: porta que precisaremos adotar em nosso servidor;
- **OPENSHIFT_NODEJS_IP**: IP do nosso servidor no OpenShift;
- **OPENSHIFT_APP_DNS**: domínio da nossa aplicação.

Nosso primeiro passo será criar o arquivo config/env/production.js, que conterá as configurações do ambiente de produção:

```
// contatooh/config/env/production.js
module.exports = {
    env: 'production',
    db: process.env.OPENSHIFT_MONGODB_DB_URL + 'contatooh',
    clientID: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    port: process.env.OPENSHIFT_NODEJS_PORT,
    address: process.env.OPENSHIFT_NODEJS_IP,
    domain: process.env.OPENSHIFT_APP_DNS
};
```

Será que isso é o suficiente? O problema é que nossa aplicação ainda não está preparada para trabalhar com as configurações port, address, domain e db. Antes de realizarmos qualquer modificação mais complexa na aplicação, vamos alterar os arquivos de configuração development.js e test.js para que façam uso dos novos parâmetros:

```
// contatooh/config/env/development.js

module.exports = {
    env: 'development',
    db: 'mongodb://localhost/contatooh',
    clientID: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    seleniumUser: process.env.SELENIUM_USER,
    seleniumUserPassword: process.env.SELENIUM_USER_PASSWORD,
    port: 3000,
    address: 'localhost',
    domain: 'localhost'
};
```

Agora o arquivo test.js:

```
// contatooh/config/env/test.js
module.exports = {
  env: 'test',
  db: 'mongodb://localhost/contatooh_test',
  clientID: process.env.CLIENT_ID,
  clientSecret: process.env.CLIENT_SECRET,
  sauceTestName: 'Contatooh E2E Testing',
  sauceUser : process.env.SAUCE_USERNAME,
  sauceKey : process.env.SAUCE_ACCESS_KEY,
  seleniumUser: process.env.SELENIUM_USER,
  seleniumUserPassword: process.env.SELENIUM_USER_PASSWORD,
  travisJobNumber: process.env.TRAVIS_JOB_NUMBER,
  travisBuild: process.env.TRAVIS_BUILD_NUMBER,
  port: 3000,
  address: 'localhost',
  domain: 'localhost'
};
```

Precisamos alterar todos os pontos da nossa aplicação onde temos esses valores fixos e substituí-los pelo valor de nosso objeto de configuração.

Modificando passport.js

Vamos começar pelo arquivo config/passport.js:

```
// código anterior comentado
module.exports = function() {

  // declarando uma variável para nos ajudar
    var githubCallback = 'http://' + config.domain + ':'
        + config.port + '/auth/github/callback';

    // código anterior omitido

  // realizando a troca
    passport.use(new GitHubStrategy({
        clientID: config.clientID,
        clientSecret: config.clientSecret,
        callbackURL : githubCallback

    // código posterior omitido
```

Quando estamos em ambiente de desenvolvimento, nosso callbackURL apontará para localhost, porém, no ambiente de produção, apontará para o domínio da aplicação. O problema é que não temos uma aplicação cadastrada no Github que aponte para este domínio. Na seção 8.2, aprendemos a cadastrar uma aplicação no GitHub, aplicação à qual delegamos o processo de autenticação de nossos usuário. Precisamos cadastrar uma nova, mas que aponte para o domínio da aplicação que será deployada no OpenShift:

Modificando Server.js

Por m, vamos alterar a linha que cria nosso servidor no arquivo server.js:

```
// contatooh/server.js

// código anterior omitido

// alterando a criação do servidor
http.createServer(app).listen(config.port, config.address,
 function(){
  console.log('Express Https Server '
    + config.address
    + ' (' + config.env
    + ') escutando na porta ' + config.port);
});
```

Experimente rodar a aplicação localmente e veri car se o Travis ainda consegue construí-la. Tudo deve continuar funcionando. Caso algum problema aconteça, veri que as alterações anteriores. É muito comum erros de digitação nesta etapa.

Estamos quase preparados para realizarmos o deploy da nossa aplicação, porém queremos que ela seja deployada pelo Travis, só assim teremos certeza de que entrará no ar uma aplicação que passou por todos os nossos testes, que são a condição *sine qua non* para o deploy contínuo. Veremos como realizar a integração entre o Travis e o OpenShi na próxima seção.

14.5 I u Z o T Z h O u S

Queremos que o Travis realize o deploy automático da nossa aplicação no OpenShi a cada commit, contanto que a construção do projeto passe por todos os testes de unidade e e2e. Até agora, editamos o arquivo .travis.yml manualmente, mas chegou a hora de instalarmos seu cliente em linha de comando, também feito em Ruby, assim como o rhc. Instalamos o cliente em linha de comando do Travis através do comando:

```
sudo gem install travis
```

Depois, precisamos estar dentro do repositório da aplicação (o que criamos no Git, não do OpenShi□!). Dentro do repositório, executaremos o comando:

```
travis setup openshift
```

O Travis fará algumas perguntas para que possa alterar automaticamente para nós o arquivo .travis.yml. O assistente perguntará o seu usuário no Openshift(email), a senha da sua conta, o nome da aplicaç ão, o domínio da sua aplicação. Perguntará também se queremos realizar o deploy apenas pelo nosso repositório no GitHub, justamente o que desejamos.

```
Shell completion not installed. Would you like to like to
install it now? |y| y
Detected repository as flaviohenriquealmeida/contatooh, is this
correct? |yes| yes
OpenShift user: seuUsuarioNoOpenShift@seuEmail.com
OpenShift password: *******
OpenShift application name: |contatooh|
OpenShift domain: contatooh-seuDominioNoOpenShift.rhcloud.com
Deploy only from usuarioGitHub/contatooh? |yes| yes
Encrypt Password? |yes| yes
```

Se editarmos nosso arquivo contatooh/.travis.yml, veremos as novas con□gurações:

```
deploy:
  provider: openshift
  user: seuUsuarioNoOpenShift@seuEmail.com
  password:
    secure: naoSePreocupeChaveEncriptadaPeloTravis
  app: contatooh
  domain: 'seuDominioNoOpenShift'
  on:
    repo: usuarioGitHub/contatooh
```

Estamos quase lá! Porém, precisamos resolver um problema antes de continuarmos. Nosso *script* Gruntfile.js cria uma cópia do projeto dentro

da pasta dist e isso nos causará problemas. O Travis não entende que esta pasta precisa ser atualizada no repositório da nossa aplicação no OpenShi.

Precisamos alterar a fase before_script do nosso arquivo .travis.yml, que deverá car assim:

```
before_script:
- node test/data.js
- grunt minifica
```

Desta vez, estamos chamando o comando grunt minifica no lugar da tarefa *default*, inclusive removemos a instrução que entrava no diretório dist. A consequência disso é que precisaremos também alterar nosso Gruntfile.js para que realize todo o processo diretamente na pasta contatooh. O desenvolvedor precisará tomar cuidado para não rodar grunt minifica em ambiente de desenvolvimento, correndo o risco de comprometer seu código. Há outras maneiras de nos prevenirmos quanto a isso, como a criação de branches, mas não entraremos em detalhes. Nosso Gruntfile.js cará um pouco mais simples. Sua versão nal cará assim:

```
module.exports = function(grunt) {
  grunt.initConfig({

    usemin : {
      html: 'app/views/**/*.ejs'
    },

    useminPrepare: {
      options: {
        root: 'public',
        dest: 'public'
      },
      html: 'app/views/**/*.ejs'
    },

    ngAnnotate: {
        scripts: {
            expand: true,
            src: ['public/js/**/*.js']
```

```
            },
        }
    });

    grunt.registerTask('minifica', ['useminPrepare',
        'ngAnnotate', 'concat', 'uglify', 'cssmin', 'usemin']);

    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.loadNpmTasks('grunt-contrib-cssmin');
    grunt.loadNpmTasks('grunt-usemin');
    grunt.loadNpmTasks('grunt-ng-annotate');
};
```

Como não usaremos mais o grunt-contrib-copy, nem o grunt-contrib-clean, podemos desinstalá-los e removê-los de nosso package.json com o comando:

```
npm uninstall grunt-contrib-copy grunt-contrib-clean
```

> Se você está começando com Git e não sabe trabalhar com *branches*, nada impede que você realize uma cópia do seu projeto localmente e execute a task grunt minifica para ter certeza de que tudo está funcionando.

Estamos no penúltimo passo! Nosso processo de mini□cação alterará os dados que o Travis trouxe de nosso repositório. Precisamos que ele as leve em consideração na fase de deploy. Para isso, existe a fase before_deploy. Nela comitaremos as alterações realizadas pelo nosso script grunt:

```
before_deploy:
  - git config --global user.email 'travis@travis-ci.com'
  - git config --global user.name 'Travis CI'
  - git checkout $TRAVIS_BRANCH
  - git add --all
  - git commit -am "Travis commit"
```

Realizamos várias mudanças em nossa aplicação para adequá-la ao OpenShi, mas como ele saberá iniciá-la? O OpenShi sempre chamará o comando npm start após o deploy ter sido realizado. Para que ele funcione, precisamos alterar nosso arquivo package.json modi cando a chave main e adicionando a chave start dentro de scripts:

```
{
  ...
  "main": "server.js",
  "scripts": {
    "start": "node server"
    "test": "echo \"Error: no test specified\" && exit 1"
  }
  ...
}
```

Pronto, chegamos ao momento nal! Vamos realizar o commit das alterações, mas antes de enviá-las para nosso repositório no Git através do comando git push, abriremos um terminal exclusivo para rodarmos o comando:

```
rhc tail contatooh
```

Lembre-se que este comando lerá continuamente o arquivo de log da sua aplicação no OpenShi.

Depois de realizar o *push* das modi cações, observe a construção de seu projeto na página do Travis. Depois de todos os testes serem executados, ele inicializará o deploy da aplicação. Como o Travis se integra facilmente com OpenShi, durante esse processo sua aplicação será parada automaticamente, os novos arquivos serão enviados e, por m, ela será inicializada, tudo isso sem sua intervenção. Basta acessar o endereço da aplicação e curtir o resultado nal:

```
https://contatooh-nomeDoSeuDominio.rhcloud.com
```

Referências Bibliográficas

[1] Dirolf Chodorow. *MongoDB: the definitive guide*. 2010.

[2] Saladage Fowler. *NoSQL Destilled, A Brief Guide to the Emerging World of Polyglot Persistence*. 2012.

[3] Valeri Karpov. □e mean stack: Mongodb, expressjs, angularjs and node.js. 2013.

[4] Oddy Osmani. Full-stack javascript with mean and yeoman. 2013.