Aplicações Web em JavaScript com Node.JS e Express

Curso JS-35

caelum

ensino e inovação

Sumário

Versão: 19.9.10

CAPÍTULO 1

COMEÇANDO O PROJETO

Ao explicar o que é Javascript para pessoas que ainda não o conhecem, é muito comum utilizarmos exempos dessa linguagem rodando no lado do cliente de uma aplicação web. Vêm à nossas cabeças exemplos envolvendo campos de formulários mostrando erros mesmo antes de enviarmos as informações para o servidor, ou funcionalidades com drag-and-drop.

Quando falamos de Node.js, estamos trazendo esse ambiente de execução de Javascript também para o lado do servidor e há diversas razões para tomarmos a decisão de utilizá-lo: as características não-blocantes durante I/O tiram um proveito bem maior da máquina do que seria natural imaginarmos -- principalmente se considerarmos que o Node.js roda sobre uma thread apenas, e com um consumo baixíssimo de processamento!

Outra vantagem, se estivermos falando de web, ainda inclui o reaproveitamento de código entre servidor e cliente, especialmente em representações de modelos e validações. E, mesmo que você vá desenvolver serviços para clientes que não rodam Javascript, a integração nativa com JSON, o formato de objetos dessa linguagem, tem se tornado o padrão de comunicação entre serviços modernos.

Nesse treinamento, você vai aprender a trabalhar com Node.js e descobrir que, além de todas as vantagens, ele é também bastante simples de entender, rápido para começarmos a produzir valor para nossos clientes, facilmente testável e extensível de forma bastante simples.

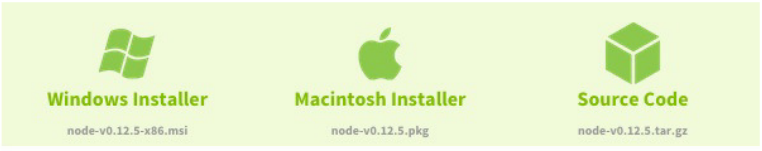
1.1 CONFIGURAÇÃO BÁSICA E CRIAÇÃO DO PROJETO

O objetivo da apostila é construir uma parte da aplicação semelhante a da própria Casa do Código. Por ser um site de e-commerce, a aplicação nos fornece a chance de implementar diversas funcionalidades e, além disso, é um domínio que não vai nos causar muitas dúvidas. Vamos começar desenvolvendo como se fosse uma aplicação web comum e depois deixaremos as operações disponíveis para serem consumidas por outros tipos de clientes, não apenas navegadores.

A primeira coisa que precisamos fazer é justamente criar e configurar o mínimo necessário para ter nossa aplicação, construída sobre o Node.js, rodando corretamente.

1.2 INSTALANDO E RODANDO O PRIMEIRO CÓDIGO

A primeira tarefa é instalar o Node.js. Esse é um processo simples: podemos simplesmente seguir os passos descritos na página de download (https://nodejs.org/download).



Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	Universal	
Mac OS X Binaries (.tar.gz)	32-bit	64-bit
Linux Binaries (.tar.gz)	32-bit	64-bit
SunOS Binaries (.tar.gz)	32-bit	64-bit
Source Code	node-v0.12.5.tar.gz	

Lá você encontra as versões do executável para cada uma das plataformas mais comumente encontradas no mercado. Se você usar algum Linux, pode optar por instalar diretamente do gerenciador de pacotes. Veja mais informações aqui: https://nodejs.org/en/download/package-manager/

Uma vez que você tem o Node.js instalado, você passa a poder chamar o comando `node` no terminal. Esse é o comando necessário para você rodar algum código javascript usando a infraestrutura fornecida pelo Node.js.

Para vermos um pouco mais de como isso funciona, começaremos com o bom e velho **Hello World**. Vamos criar um simples arquivo chamado **hello-world.js** que mostra uma mensagem:

```
console.log("Oi mundo com Node.js");
```

Salve o arquivo e abra o terminal (também chamado de *console*). Para executar, basta entrarmos na pasta onde salvamos o arquivo e usar o comando que roda o Node:

```
node hello-world.js
```

Como esperado, você deve ter visto a mensagem impressa no próprio console.

1.3 EXERCÍCIOS: SUA VEZ DE TESTAR O NODE

1. Crie um arquivo chamado `hello-world.js` na pasta raiz de seu usuário. Imprima uma simples mensagem no console dentro desse arquivo:

    ```
    console.log("Olá mundo com Node!");
    ```

2. Abra uma nova janela do terminal, vá até a pasta onde você salvou o arquivo e execute o Node, indicando para o arquivo que criamos:

    ```
    node hello-world.js
    ```

3. Verifique se a mensagem foi impressa corretamente no terminal.

1.4 DO NAVEGADOR AO BACK-END

Historicamente, quando pensamos em usar Javascript, naturalmente importamos o código Javascript em uma página HTML e o próprio navegador interpretará o código. Nesse cenário, o `console.log` mostraria a mensagem apenas para os desenvolvedores que estivessem com um console aberto (F12). O navegador é o interpretador mais comum de JavaScript que conhecemos!

O Node, no entanto, tem a proposta de rodar códigos Javascript diretamente no servidor, usando um ambiente de execução próprio, capaz de interpretar a linguagem Javascript e oferecer bibliotecas para permitir desenvolvimento do *back-end* das aplicações. Com o Node, o Javascript fez um movimento de migração do cliente (navegador) para o servidor.

Esse processo não é nada novo. A linguagem Java, que é muito por famosa por seu uso em aplicações no servidor, começou a ser conhecida, lá na década de 90, justamente para prover mais poder e possibilidades dentro do navegador. Talvez você se lembre das famosas *applets*, muito usadas para criar interações mais ricas e jogos de navegador. Até hoje, encontramos aplicabilidade para elas, por exemplo para casos em que precisamos usar o certificado digital para acessar informações sobre empresas ou em alguns teclados virtuais de bancos.

Quando temos que usar uma applet somos obrigados a instalar uma tal de *Java Runtime Environment* e seu plugin para o navegador, justamente o ambiente necessário para que se possa executar um código Java. A mesma coisa acontece quando escrevemos um código Javascript dentro de uma página.

É necessário ter um ambiente que possa interpretar aquele script e executá-lo da forma adequada. A diferença é que esses ambientes já vem integrados com os navegadores que usamos e, por isso, não somos obrigados a instalar nada a mais. Abaixo temos os exemplos das *engines* de javascript que vêm nos

navegadores mais comuns:

- Chrome: V8(https://code.google.com/p/v8/)
- Firefox: SpiderMonkey(https://developer.mozilla.org/pt-BR/docs/Mozilla/Projects/SpiderMonkey)
- Internet Explorer: Chakra(http://blogs.msdn.com/b/ie/archive/2012/06/13/advances-in-javascript-performance-in-ie10-and-windows-8.aspx)

Além dessas, outra *engine* de Javascript que já está famosa é a *Nashorn*, que vem embutida no Java 8. Ou seja, já é possível executar pedaços de código Javascript a partir de aplicações Java.

Node.js e a V8

No navegador é muito comum você usar recursos em JS para controlar melhor a sua página. Algumas situações são:

- escutar eventos de cliques de botões;
- buscar por elementos dentro da página;
- executar validações das entradas dos campos do formulário;

Já quando você for rodar seu código Javascript dentro do Node.js, você precisaria de outro conjunto de bibliotecas, que te dará a possibilidade de escrever códigos para fazer conexões com o banco de dados, acesso a arquivos, chamadas no sistema operacional, tratar requisições e respostas, etc.

O que o Node.js fez foi construir uma plataforma de desenvolvimento baseada na *engine* de Javascript do Google, sem as funcionalidades específicas de navegadores, que o Chrome adiciona, e adicionando funcionalidades importantes para desenvolvimento *back-end*.

O mundo vem usando o Node.js para escrever aplicações no lado do servidor que são capazes de atender muitas requisições sem muita perda de velocidade nas respostas. Ou seja, aplicações que mantém o requisito de performance mesmo quando estão sob uma demanda forte de uso. Esse requisito também é conhecido como **escalabilidade**.

Você perceberá durante o desenvolvimento da aplicação que o Node.js nos impõe um modelo de programação um tanto desconfortável para quem está habituado a pensar em desenvolvimento síncrono, a princípio. Durante o treinamento, no entanto, nos acostumaremos com esse modo e entenderemos o valor de trabalhar com esse paradigma.

1.5 ATENDENDO À PRIMEIRA REQUISIÇÃO

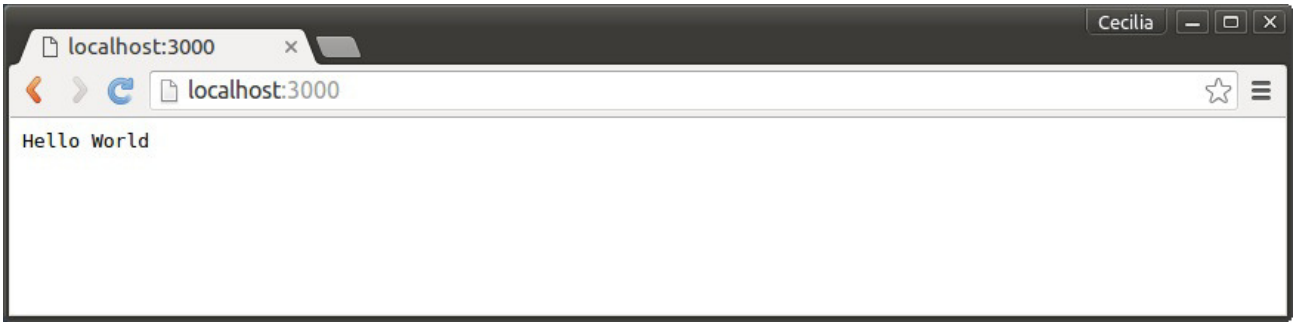
Embora criar scripts para rodar no terminal seja uma utilização válida para o Node.js, ela não é a mais usual. A grande maioria do desenvolvimento com Node.js, é direcionada à web. Vamos, então, começar a atender nossa primeira requisição.

Usualmente, chamamos de servidor quem cuida de requisições e respostas. Assim, precisaremos subir um servidor *http* e gerar uma resposta para o cliente. O Node tem uma biblioteca disponível para isso, que utilizaremos no arquivo **ola-server.js**:

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

var ip = "localhost";
var porta = 3000;

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(porta, ip);

console.log("Servidor rodando em http://"+ip+":"+porta);
```

Ao rodar esse script, indo no terminal e digitando `node ola-server`, ficamos com o terminal travado, já que acabamos de levantar um servidor e ele deve ficar esperando por novas requisições. Agora podemos ir até o navegador e acessar endereço IP e porta especificados.

Perceba que passar a extensão `.js` no comando é opcional, isto é, daria na mesma se tivéssemos chamado: `node ola-server.js`.



Entendendo o código

A primeira coisa que precisamos foi da biblioteca responsável por todo o tratamento do protocolo **HTTP**.

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

A função *require* é responsável por carregar as bibliotecas que estão disponíveis no Node.js. Ela procura por um arquivo Javascript com o mesmo nome do parâmetro. A biblioteca `http` que acabamos de usar já vem por padrão na plataforma, e, mais pra frente, ainda vamos usar algumas outras que precisarão ser instaladas.

Se tiver curiosidade sobre o que mais está disponível, além de buscar a documentação do Node, você ainda pode entrar no diretório onde o Node foi instalado na sua máquina e olhar os arquivos `.js` que estão nele, na pasta `lib` (se existir), e em outra pasta chamada `node_modules`.

A função *require* segue a especificação **CommonJS**, que é justamente uma tentativa de padronização de carregamento de libs em ambientes Javascript. Um pouco mais sobre ela pode ser encontrada na página http://www.commonjs.org/.

Agora que carregamos o módulo, podemos acessar tudo que está disponível dentro dele. E aí vem a segunda parte do código:

```
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(porta, ip);
```

O método `createServer`, como o próprio nome diz, serve para criarmos e configurarmos um novo objeto que representa um servidor. Daí, chamamos a função `listen` para indicar qual porta e endereço IP esse servidor vai monitorar. Essa é a parte simples.

A configuração do servidor, contudo, é a parte um pouco mais complicada. Note que foi necessário fornecer uma função que será responsável por tratar todas as requisições que chegarem no sistema pelo IP e porta indicados. Essa função recebe dois parâmetros:

- o primeiro (`req`) representa a requisição em si e podemos usá-lo, por exemplo, para acessar

os parâmetros enviados pelo usuário.

- o segundo (`res`) é o objeto que possibilita escrever a resposta para o cliente.

No nosso exemplo estamos escrevendo um simples texto, mas nada nos impede de escrever um HTML mais complexo.

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

var ip = "localhost";
var porta = 3000;

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end("<html><body>
        <a href='http://www.caelum.com.br'>Site da caelum</a></body></html>");
}).listen(porta, ip);

console.log("Servidor rodando em http://"+ip+":"+porta);
```

Note que tivemos que mudar o cabeçalho chamado `Content-Type` para *text/**html***, para que o cliente (nesse caso, o navegador) entenda que queremos que o conteúdo seja exibido como uma página. Vamos ver sobre mais esse *header* quando entrarmos na discussão sobre integrações.

Você também vai perceber que o tempo todo trabalharemos passando funções como argumento. Isso se deve ao modelo de programação imposto pelas características de I/O não-blocantes do Node.js, que também será amplamente discutido durante o nosso treinamento.

1.6 EXERCÍCIOS: PRIMEIRO SERVIÇO COM NODE

1. Crie um arquivo chamado `ola-server.js` na pasta raiz de seu usuário:

2. Dentro do arquivo `ola-server.js` , criaremos o nosso servidor:

```
var http = require('http');
var porta = 3000;
var ip = "localhost";

http.createServer(function (req, res) {
    console.log('Recebendo request!');
    res.end();
}).listen(porta, ip);

console.log("Servidor executando em http://" + ip + ":" + porta);
```

3. Inicie o servidor pelo terminal:

```
node ola-server
```

4. Observe que a mensagem *"Servidor executando em http://localhost:3000/"* aparece assim que o node carrega o arquivo.

5. Agora, abra o navegador e acesse a aplicação em **http://localhost:3000/**. Como não escrevemos nada

na resposta, a tela aparece em branco, mas note que a mensagem "Recebendo request!"_aparece no terminal toda vez que você acessa ou recarrega a página!

6. Até agora, a chamada a `res.end()` não está recebendo parâmetro e, por isso, nada está sendo devolvido para o navegador. Vamos mudar isso escrevendo um html bem básico para ser retornado e descrevendo o Content-Type de acordo:

```
...

http.createServer(function (req, res) {
    console.log('Recebendo request!');
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end("<html><body>Uma mensagem na tela!</body></html>");
}).listen(porta, ip);

...
```

7. Acesse novamente a aplicação no navegador e veja que a mensagem realmente aparece na tela.

1.7 COMEÇANDO COM O EXPRESS

No exemplo anterior retornamos um código HTML que deveria ser renderizado pelo navegador. Mesmo que nosso exemplo tenha sido muito simples, já podemos perceber que escrever uma página inteira ali seria muito complexo e prejudicaria bastante a visualização e a manutenibilidade da página.

Além disso, até agora estamos simplesmente acessando o endereço raiz da aplicação, mas em um cenário real vamos querer que urls diferentes respondam conteúdos diferentes. Nesse cenário, poderíamos ter:

```
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  if(req.url ==  "/produtos") {
        res.end("<html><body>Listando produtos</body></html>");
  } else {
        res.end("<html><body>Outra url</body></html>");
  }

}).listen(porta, ip);
```

Esse conjunto de tarefas faz com que o esforço para construir uma aplicação web, usando diretamente as API's do Node.js, seja muito grande. Para conseguirmos atingir esse objetivo vamos começar a usar uma outra biblioteca escrita em Javascript chamada **Express**.

O Express é um framework web compatível com as API's fornecidas pelo Node.js. A ideia é nos fornecer uma estrutura mínima para criarmos uma aplicação web que suporte algumas características básicas, como as que foram citadas acima. Vamos lembrar delas e de mais algumas:

- isolar o código de tratamento de diferentes urls
- isolar a escrita do código html da lógica da aplicação
- responder conteúdos com formatos diferentes, baseado nas informações do request

- lidar com os métodos de envio de dados, por exemplo requisições feitas com GET e com POST

Criando um novo projeto com o Express

Até agora, fizemos tudo na mão. A partir desse momento, no entanto, vamos ser obrigados a adicionar bibliotecas extras. O Node.js procura por essas bibliotecas em pastas específicas e, além disso, muitas vezes uma biblioteca depende de outras, nos obrigando a realizar mais de um download.

Um outro ponto que devemos nos preocupar é com as atualizações dessas libs, já que muitas vezes atualizar implica em atualizar outras bibliotecas também. Para não termos que lidar com este tipo de detalhe, vamos usar uma ferramenta chamada **npm**, que é o acrônimo para *Node Package Manager*. O **npm** é instalado no mesmo instante que instalamos o Node.js. Podemos acessá-lo pelo terminal através do comando `npm` .

Vamos criar uma pasta que vai conter os arquivos do nosso projeto e entrar nela. Então, usaremos o próprio `npm` para transformar nossa pasta em um projeto gerenciado pelo `npm` e criar a estrutura básica que contém as informações do nosso projeto.

```
mkdir casadocodigo
cd casadocodigo
npm init
```

O comando vai pedir que você preencha várias informações pertinentes ao projeto, como nome do projeto, versão atual, autor, licensa, repositório, etc. Também é possível deixar todas as opções com valores padrão e voltar, mais tarde, para preencher o que não sabíamos a princípio!

Esse comando cria um arquivo chamado *package.json* que contém informações básicas sobre seu projeto. Algo como:

```
{
    "name": "casadocodigo",
    "version": "1.0.0",
    "description": "Projeto usado para o ensino de Node.js na Caelum",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "keywords": [
        "Node",
        "ensino",
        "JS-35",
        "Caelum"
    ],
    "author": "",
    "license": "ISC"
}
```

Este arquivo vai ser automaticamente alterado durante o projeto, quando começarmos a adicionar novas bibliotecas. A primeira que adicionaremos é o próprio **Express** e, para isso, vamos usar o npm novamente.

```
npm install express --save
```

O comando *install* vai realizar o download dos arquivos e colocá-los numa pasta chamada *node_modules*, dentro da própria pasta do seu projeto. A opção *--save* é necessária para que o próprio npm altere o nosso arquivo *package.json* e já deixe explícito que o projeto possui essa dependência.

```
{
    "name": "casadocodigo",
    ...
    "dependencies": {
        "express": "^4.13.4"
    }
}
```

Esse parâmetro é bem importante. Em geral, várias pessoas podem trabalhar em um mesmo projeto e, caso não fosse forçado que a dependência ficasse no *packages.json*, correria o risco de uma pessoa do time baixar o projeto e não saber quais libs ela deveria instalar. Agora quando um novo integrante do time baixar o projeto para sua máquina, só vai ser necessário que ele rode `npm install`.

1.8 EXERCÍCIOS: INICIANDO O PROJETO COM NPM

1. Do terminal, vá para a pasta raiz do seu usuário, crie uma pasta `casadocodigo` e entre nela.:

   ```
   cd
   mkdir casadocodigo
   cd casadocodigo
   ```

2. Inicialize o projeto usando o `npm`:

   ```
   npm init
   ```

3. Conforme o comando perguntar as opções do projeto, apenas dê *enter*, deixando todos os valores padrão. No passo final da criação, ele deve mostrar o JSON de configuração do projeto, que ficará em `package.json`. Deve ser bastante parecido com esse:

   ```
   {
   ```

```
    "name": "casadocodigo",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC"
}
```

4. Agora que temos o esqueleto do projeto pronto, já podemos instalar o *Express*, lembrando de colocar a opção `--save` no comando, para que o npm já anote a dependência no *package.json*:

```
npm install express --save
```

5. Para confirmar que está tudo certo, verifique se as dependências foram adicionadas ao *package.json* usando o comando `cat package.json`. O resultado deve ser parecido com:

```
{
    "name": "casadocodigo",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
        "express": "^4.13.4"
    }
}
```

1.9 ATENDENDO A PRIMEIRA REQUISIÇÃO COM O EXPRESS

Com o **Express** baixado e tudo configurado, podemos começar a implementar nosso projeto. A nossa primeira tarefa é realizar a listagem dos produtos da nossa loja. O endereço **/produtos** deve ser o responsável por essa ação.

Nesse exato momento, se tentarmos digitar o endereço *http://localhost:3000/produtos* vamos receber uma mensagem de erro do navegador dizendo que não encontrou nada. O que faz todo sentido, dado que não subimos nenhum servidor. Para começar a resolver isso, vamos usar o Express para criar um servidor numa porta específica. Criaremos um arquivo chamado **app.js** na raiz do nosso projeto.

```
var express = require('express');
var app = express();

var server = app.listen(3000, function () {
    console.log("Servidor rodando");
});
```

A invocação do `require("express")` retorna uma função que é guardada na variável *express*. Para realmente iniciarmos o *express* precisamos chamar essa função e, é exatamente por isso, que fazemos

`express()`. Caso você tenha ficado curioso, adicione um `console.log(express.name)` e você verá que o retorno será o nome da função que, no caso, é *createApplication*. Esta função retorna um objeto que literalmente representa todo o contexto do express e é ele que vamos usar sempre que quisermos nos comunicar com o framework. Por exemplo, invocamos o método *listen* para subir o servidor. Esse código não tem nenhuma mágica, simplesmente encapsula o mesmo código que tínhamos feito quando estávamos usando a API do Node.js diretamente.

Agora, quando tentamos acessar o mesmo endereço pelo navegador, recebemos a mensagem **Cannot GET /produtos**. A requisição até chegou no express, mas ele não conseguiu descobrir qual função deveria tratar essa url. Para resolvermos isso, vamos usar um outro método chamado *get*.

```
var express = require('express');
var app = express();

app.get('/produtos', function (req, res) {
    res.send("listagem de produtos");
});

var server = app.listen(3000, function () {
    console.log("Servidor rodando");
});
```

O método *get* recebemos o endereço que queremos tratar e também a função responsável por executar a lógica relativa a requisição. Perceba que ela também recebe um *request* e um *response*, do mesmo jeito que já fizemos mais cedo. Por fim, invocamos o método *send* para enviar um texto para o navegador. Nesse exato momento, se reiniciarmos o servidor e tentarmos acessar a mesma url, vamos receber o resultado esperado.

1.10 EXERCÍCIOS: ATENDENDO A PRIMEIRA REQUISIÇÃO COM O EXPRESS

1. Crie um arquivo chamado `app.js` no diretório da nossa aplicação e, dentro dele, inicie o `express`:

    ```
    var express = require('express');
    var app = express();

    var server = app.listen(3000, function () {
        console.log('Servidor rodando');
    });
    ```

2. Criamos a configuração mínima de nosso servidor, mas ainda não registramos nenhuma url em nossa aplicação. Vamos registrar um tratamento para a url `/produtos`, retornando uma mensagem simples para o usuários e fazendo um log toda vez que uma requisição for recebida:

    ```
    var express = require('express');
    var app = express();

    app.get('/produtos', function (req, res) {
        console.log('Recebeu requisição');
    ```

```
    res.send('<h1>Listagem de produtos</h1>');
});

var server = app.listen(3000, function () {
    console.log('Servidor rodando');
});
```

3. Inicie o servidor e nossa aplicação:

```
node app
```

4. Acesse o endereço `http://localhost:3000/produtos` e veja se a mensagem é exibida no navegador.

1.11 EXIBINDO PÁGINAS HTML COM O EJS

Para fechar, não queremos ter que escrever o HTML dentro da nossa lógica. Para resolver este problema vamos usar um projeto que já possui integração com Express, chamado **EJS**, que significa *Embedded JavaScript*. Ele nos permite escrever páginas dinâmicas, o que é muito útil para qualquer aplicação web. Para adicionarmos o EJS no nosso projeto usaremos o npm.

```
npm install ejs --save
```

Agora precisamos alterar nosso código para ensinar o Express que ele deve usar o EJS como mecanismo de template para renderizar nossas páginas.

```
var express = require('express');
var app = express();

app.set('view engine', 'ejs');

app.get('/produtos', function (req, res) {
    console.log('Requisição chegou');
    res.send('<h1>Listagem de produtos</h1>');
});
```

```
app.listen(3000, function () {
    console.log('Servidor rodando');
});
```

O método *set* serve para criar uma nova variável dentro do express. Nesse caso estamos passando a string "view engine" que é usada internamente pelo framework para buscar o módulo responsável por renderizar as views. Além disso já alteramos a função que atende o request e passamos a invocar a função *render*. O argumento é justamente o caminho do arquivo que contém nosso HTML. Só que quando acessamos esse endereço, recebemos o seguinte erro.

```
Error: Failed to lookup view "produtos/lista" in views directory
"~/ambiente/desenvolvimento/javascript/casadocodigo/views"
```

Perceba que foi buscado pela nossa página a partir da pasta chamada **views** que deveria existir na raiz da nossa aplicação. Para tudo funcionar, basta que criemos a página na localização correta. Vamos criar o arquivo **lista.ejs** dentro da pasta *views/produtos*.

```
<html>
    <body>
        Aqui listaremos os produtos
    </body>
</html>
```

Pronto! Quando acessarmos o endereço **/produtos** vamos receber como resposta o nosso HTML.

Esse processo inicial é muito importante. Fizemos apenas o básico necessário para rodar uma aplicação com o express, mas os conceitos vistos aqui serão necessários durante todo o treinamento.

1.12 EXERCÍCIOS: CRIANDO UMA PÁGINA COM O EJS

1. No diretório do nosso projeto, instale o `ejs`:

    ```
    npm install ejs --save
    ```

2. Agora vamos configurar o express para usar o `ejs` como renderizador de páginas. Abra o arquivo `app.js` e adicione o código abaixo log após a declaração do `express`:

    ```
    var express = require('express');
    var app = express();

    app.set('view engine', 'ejs'); //linha nova
    ...
    ```

3. Dentro do projeto, crie o diretório **views** e, dentro dele, crie o diretório **produtos**. Você pode fazer isso pela interface gráfica do sistema operacional ou, pelo terminal, com o comando:

    ```
    mkdir -p views/produtos
    cd views/produtos
    ```

4. Crie uma página chamada `lista.ejs` dentro do diretório **views/produtos**. O conteúdo do arquivo é um HTML normal:

```
<html>
    <body>
        <h1>Aqui listaremos os produtos</h1>
    </body>
</html>
```

5. Por fim, faça com que ao acessar `/produtos` seja exibida a página que acabamos de criar. Dentro do arquivo `app.js`, modifique a linha dentro da configuração da rota para exibir a página em vez da mensagem:

```
...
app.get('/produtos', function (req, res) {
    console.log('Requisição chegou');
    res.render('produtos/lista');    // linha alterada
});
...
```

6. Reinicie o servidor com `node app` e acesse **http://localhost:3000/produtos** e veja se tudo ocorreu como esperado.

CAPÍTULO 2

LISTANDO OS PRODUTOS

Agora que já temos um início para nossa aplicação e já conseguimos mostrar uma versão inicial da tela de listagem de produtos, já dá pra começarmos a enxergar a estrutura de uma aplicação em Node.

Antes de produzirmos ainda mais código, vamos investir um tempo na organização do projeto em pastas com propósitos mais específicos e arquivos mais coesos.

2.1 MELHORANDO A ESTRUTURA DO PROJETO

No momento, temos apenas um arquivo chamado de **app.js**, que contém todo o código do nosso projeto. Ainda que ele seja um arquivo pequeno agora, ele tenderá a crescer conforme evoluimos com o projeto e as configurações e controladores se acumulam.

Ele também tem mais de uma responsabilidade: além de subir o servidor na porta definida, ele também recebe a configuração do express, da *view engine* escolhida e sabe o que responder a uma requisição feita por um cliente. Vamos separar essas partes em arquivos diferentes e mais específicos.

Começaremos extraindo para um arquivo `server.js` apenas o que é relativo a subir o servidor na porta correta. Para isso, consideraremos que todo o resto da informação que, no momento, está no nosso `app.js` esteja em outro arquivo chamado `custom-express.js` . Se separarmos apenas essa parte do código, teremos algo tão simples quanto:

```
var customExpress = require('./custom-express');
var app = customExpress();

app.listen(3000, function () {
    console.log('Servidor rodando');
});
```

Toda a configuração do express e das rotas foi jogada para um arquivo que precisaremos criar, chamado `custom-express.js` que também ficará dentro da pasta raiz do projeto. O "." na frente serve para indicar que o caminho descrito parte da pasta atual. Perceba que a função *require* é utilizada para carregar qualquer tipo de arquivo JavaScript do seu projeto, seja ele um módulo padrão do Node.js, uma biblioteca que instalamos ou um arquivo nosso.

Ela já é esperta o suficiente para entender que, quando passamos apenas o nome de um módulo, queremos algum padrão ou instalado, e que, quando passamos um caminho direto para um arquivo, queremos que ela utilize um arquivo nosso.

A função `require` funciona assim: ela busca o arquivo JS referente ao que você passou como parâmetro e devolve o que quer que esteja na variável `module.exports`. Como qualquer outra variável do Javascript, essa variável pode guardar todo tipo de conteúdo: valores, objetos JSON e, mais usualmente, uma função.

PARA SABER MAIS: CHAMANDO A FUNÇÃO *INLINE*

Perceba que, no `server.js`, pegamos o retorno do *require* e, na linha seguinte, invocamos a função sem nenhum argumento. Também poderíamos fazer isso em uma linha só, chamando a função com () logo após o `require`. No `server.js` poderíamos, assim, substituir as duas primeiras linhas por:

```
var app = require('custom-express')();
```

Como queremos que o `custom-express.js` seja chamado via `require`, precisamos que ele coloque informações na variável `module.exports`. E porque precisamos disponibilizar uma instância do Express completamente configurada, o código para alcançar esse objetivo colocaremos uma função que, ao final da sua execução, vai prover uma instância do app:

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

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

    app.set('view engine', 'ejs');

    app.get('/produtos', function (req, res) {
        console.log('Requisição chegou');
        res.render('produtos/lista');
    });
};
```

O seu arquivo pode declarar quantas variáveis e funções você quiser, mas a única coisa que fica acessível para quem o carregou com o *require* é o que foi atribuido ao `module.exports`. No nosso caso, atribuimos uma função e, por isso, temos a opção de invocá-la.

Essa variável está disponível em cada arquivo que é carregado baseado na especificação **CommonJS**, que é justamente o padrão usado pelo Node.js.

Isolando os controladores da aplicação

Estamos em um bom caminho para a separação de responsabilidades, mas nossa versão do `custom-express` ainda tem duas responsabilidades: a configuração do express e o código que será executado quando um cliente fizer uma requisição para a URL `/produtos`.

Se pensarmos em um projeto, ainda que de pequeno porte, teremos usualmente cerca de quatro rotas para cada modelo: criação, alteração, remoção e listagem. É fácil notar que o custom-express logo ficará gigantesco -- e códigos grandes tendem a se tornar difíceis de dar manutenção, além dos problemas com conflitos em sistemas de controle de versão.

A separação mais comum para quem trabalha com MVC atualmente, costuma ser a de criar um arquivo por modelo, que cuidará das rotas e códigos de *controller* deles. Na literatura sobre Node, a nomenclatura para a pasta que contém esses códigos pode ser **routes** ou **controllers**. Utilizaremos a primeira no treinamento.

```
casadocodigo/
        custom-express.js
        server.js

        routes/
            produtos.js
            outro-modelo.js
        views/
            produtos/
                lista.ejs
                outra-pagina-de-produtos.ejs
            outro-modelo/
                paginas-do-outro-modelo.ejs
        node_modules/
            modulos-instalados/
```

No arquivo `routes/produtos.js`, precisaremos da variável que representa o Express, já com a *view engine* configurada, então passaremos sua instância como parâmetro da função que ficará acessível para ser chamada do **custom-express.js**.

```
    module.exports = function(app) {
        app.get("/produtos",function(req,res) {
            res.render('produtos/lista');
        });
    }
```

Exportamos uma função, que quando invocada, vai ter a responsabilidade de mapear as rotas da aplicação para suas respectivas funções de tratamento, também chamada de **controllers**. Já que o objetivo delas vai ser o de receber os dados da requisição, processá-los e depois retornar uma resposta para o cliente.

Para finalizar, precisamos que essa função seja invocada depois da configuração do EJS no nosso Express. Para isso, vamos pedir para o `custom-express` carregar o módulo que cuida das rotas de produto, passando a instância do `app` para ele:

```
var app = require('express');

module.exports = function () {
    app.set('view engine', 'ejs');

    require('./routes/produtos')(app);  // pede a função e já a executa

    return app;
```

```
};
```

Como a função exportada a partir do arquivo de rotas recebe a referência para o objeto que representa o *express*, somos obrigados a passá-lo como argumento.

2.2 EXERCÍCIOS: ACERTANDO A ESTRUTURA DA APLICAÇÃO

1. Abra a pasta `casadocodigo` . Dentro dela crie uma nova pasta chamada **routes**, além da pasta **views**, que já existe. Você pode fazer isso pela interface gráfica, se preferir. No terminal, no entanto, ficaria assim:

```
cd casadocodigo
mkdir routes
```

2. Começaremos a separar as responsabilidades da aplicação em arquivos diferentes. Dentro da pasta **routes**, crie o arquivo chamado **produto.js**:

```
module.exports = function (app) {
    app.get("/produtos",function(req,res) {
        res.render('produtos/lista');
    });
};
```

3. Isole a configuração do express. Crie um arquivo na pasta `casadocodigo` chamado **custom-express.js** e mova a configuração do framework e o require da rota recém-criada para lá:

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

module.exports = function () {
    var app = express();
    app.set('view engine', 'ejs');

    require('./routes/produtos')(app);

    return app;
};
```

4. Isole, agora, a configuração para subir o servidor em um novo arquivo que chamaremos de **server.js**:

```
var app = require('./custom-express')();

app.listen(3000, function () {
    console.log('Servidor rodando');
});
```

5. Suba o servidor usando `node server` e acesse **http://localhost:3000/produtos**. Veja se tudo está funcionando como antes.
6. Remova o arquivo `app.js`, que acumulava todas responsabilidades até então e não é mais necessário -- e, portanto, não será mais usado.

2.3 ACESSANDO O BANCO DE DADOS

Os dados dos produtos ficarão guardados em um banco de dados, no nosso caso o MySQL. Então nossa primeira tarefa é justamente criar a tabela e inserir alguns produtos. Abaixo temos o script de criação da tabela de produtos que precisamos.

```
CREATE TABLE livros (
  id int(11) NOT NULL AUTO_INCREMENT,
  titulo varchar(255) DEFAULT NULL,
  descricao text,
  preco decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (id)
);
```

Para não termos que implementar a tela de cadastro nesse momento, já vamos também inserir alguns produtos.

```
insert into livros(titulo,descricao,preco)
values('Comecando com nodejs','livro introdutorio sobre nodejs',39.90);

insert into livros(titulo,descricao,preco)
values('Comecando com javascript','livro introdutorio sobre javascript'
,39.90);

insert into livros(titulo,descricao,preco)
values('Comecando com express','livro introdutorio sobre express',39.90);
```

Agora que já temos a tabela criada e alguns produtos adicionados, chegou a hora de realizarmos a query. Assim como em outras linguagens, não queremos ter que lidar com detalhes do protocolo do banco de dados, independende de qual seja. Para resolver este problema a comunidade Javascript, em volta do Node.js, já desenvolveu um driver de acesso ao MySQL. Assim como foi com o *express*, precisamos adicioná-lo ao nosso projeto para conseguir usá-lo.

```
npm install mysql --save
```

Com o *driver* instalado, podemos carregar o módulo e realizar a nossa query. Vamos fazer dentro da função responsável, no arquivo **routes/produtos.js**.

```
app.get("/produtos",function(req,res) {
```

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

    var connection = mysql.createConnection({
        host     : 'localhost',
        user     : 'root',
        password : '',
        database : 'casadocodigo'
    });


    connection.query('select * from livros',
        function(err, result,fields) {
            res.send(result);
        }
    );

    connection.end();
});
```

Não fizemos nada que já não tenha sido estudado antes. A função *createConnection* espera um JSON como argumento, com as informações relativas a sua conexão. Com a conexão na mão invocamos a função *query* que recebe a consulta e uma função que será invocada quando o driver obtiver os dados do banco de dados, também conhecida como **callback**. Aqui vale a pena pararmos para dar uma olhada nos argumentos recebidos no **callback**.

- err; argumento que contém informações sobre possíveis erros da consulta. Deve ser verificado para se precaver de qualquer falha na execução da query
- result; JSON com o resultado do **select** realizado
- fields; JSON com informação sobre cada coluna retornada na consulta.

Perceba que nosso *callback* usa o *response* para enviar o resultado para o nosso navegador. Quando acessar a url de produtos, o resultado esperado deve ser algo parecido com o que segue:

```
[
    {
        "id": 13,
        "titulo": "Comecando com nodejs",
        "descricao": "livro introdutorio sobre nodejs",
        "preco": 39.90
    },
    {
        "id": 14,
        "titulo": "Comecando com javascript",
        "descricao": "livro introdutorio sobre javascript",
        "preco": 39.90
    },
    {
        "id": 15,
        "titulo": "Comecando com express",
        "descricao": "livro introdutorio sobre express"
        "preco": 39.90
    }
]
```

O driver já transforma o resultado da query em um JSON, pois já é mais do que conhecido que este é

o formato ideal para representar estruturas no mundo Javascript. Para não deixar conexões abertas com o nosso banco de dados, fechamos a nossa no fim da função de listagem de produtos.

Um pensamento que pode estar atravessando sua mente é o motivo que nos levou a usar o **require** de dentro da função. Na verdade não existe nenhuma regra que nos obriga a carregar um módulo no inicio do arquivo, ainda mais se ele for ser usado apenas em alguma função específica. No nosso caso, por enquanto, só queremos o MySQL para listar os produtos, então faz todo sentido deixar essa chamada perto do restante do código.

Listando os produtos retornados na páginas

Estamos mandando um JSON para a página, mas o que realmente precisamos é montar um HTML exibindo os dados de cada um dos produtos. O *EJS* nos permite escrever trechos de códigos dinâmicos dentro das nossas páginas. Por exemplo, para fazer a listagem podemos escrever o seguinte código dentro do arquivo *views/produtos/lista.ejs*.

```
<html>
    <body>
        <table>
            <% for(var i=0; i<lista.length; i++) {%>
                <tr>
                    <td><%= lista[i].titulo %></td>
                    <td><%= lista[i].preco %></td>
                    <td><p><%= lista[i].descricao%></p></td>
                </tr>
            <% } %>
        </table>
    </body>
</html>
```

Simplesmente usamos a sintaxe do JavaScript dentro de uma página HTML. Como colocamos os trechos dinâmicos dentro de **<% ... %>**, o próprio EJS reconhece tais trechos e executa o código dinâmico. Agora devemos alterar o código do nosso controller, para que ele tente renderizar a página, ao invés de enviar diretamente o JSON.

```
connection.query('select * from livros',
    function(err, result,fields) {
        res.render("produtos/lista");
    }
);
```

Uma pergunta que ficou é: Como a variável **lista** apareceu no script? Na verdade, se acessarmos a listagem de produtos vamos receber o seguinte erro.

```
ReferenceError: /Users/ambiente/desenvolvimento/javascript/
    casadocodigo/views/produtos/lista.ejs:4
   2| <body>
   3| <table>
>> 4|     <% for(var i=0; i<lista.length; i++) {%>
   5|     <tr>
   6|         <td><%= lista[i].titulo %></td>
   7|         <td><%= lista[i].preco %></td>
```

```
lista is not defined
```

Realmente o EJS não encontrou nenhuma variável com esse nome quando foi tentar executar o código JavaScript acima. Para que este trecho funcione, somos obrigados a passar um parâmetro com o mesmo nome para a página.

```
connection.query('select * from livros',
    function(err, result,fields) {
        res.render("produtos/lista",{lista:result});
    }
);
```

Simplesmente passamos um JSON onde a chave é o nome da variável que será utilizada na view. Já fique acostumado, é muito comum API's do JavaScript usarem um JSON quando uma função pode receber um número qualquer de argumentos. Por exemplo, aqui poderíamos passar quantas variáveis a gente quisesse para a página. A mesma coisa aconteceu quando usamos a função *createConnection*.

```
    var connection = mysql.createConnection({
        host     : 'localhost',
        user     : 'root',
        password : '',
        database : 'casadocodigo'
    });
```

Além das opções mais comuns, podemos passar um *timeout* para a criação da conexão.

2.4 EXERCÍCIOS: EXIBINDO OS PRODUTOS NA PÁGINA

1. Abra um terminal e acesse o mysql:

```
mysql -u root
```

Crie um banco de dados chamado **casadocodigo**, e conecte nele:

```
create database casadocodigo;
use casadocodigo;
```

2. Crie a tabela para armazenar os livros:

```
CREATE TABLE livros (
  id int(11) NOT NULL AUTO_INCREMENT,
  titulo  varchar(255) DEFAULT NULL,
  descricao  text,
  preco  decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (id)
);
```

3. Insira alguns livros:

```
insert into livros(titulo,descricao,preco)
values('Começando com nodejs','Livro introdutório sobre nodejs',39.90);

insert into livros(titulo,descricao,preco)
values('Começando com javascript','Livro introdutório sobre javascript'
,39.90);
```

```
insert into livros(titulo,descricao,preco)
values('Começando com express','Livro introdutório sobre express',39.90);
```

4. Com o banco pronto e com dados, começaremos a mexer em nossa aplicação. Instale o módulo do mysql. No diretório da aplicação, rode o comando:

```
npm install mysql --save
```

5. Abra o arquivo **routes/produtos.js**, e remova o conteúdo do controller associado a rota de listagem.

```
module.exports = function (app) {
    app.get("/produtos",function(req,res) {

    });
};
```

6. Agora inclua a abertura da conexão e a consulta no banco, retornando o resultado para o navegador. Não esqueça de fechar a conexão no final do método:

```
app.get("/produtos",function(req,res) {
    var mysql  = require('mysql');

    var connection = mysql.createConnection({
        host     : 'localhost',
        user     : 'root',
        password : '',
        database : 'casadocodigo'
    });

    connection.query('select * from livros',
        function(err, result,fields) {
            res.send(result);
        }
    );

    connection.end();
});
```

7. Acesse o navegador e acesse `http://localhost:3000/produtos` e veja a lista de livros.

8. Vamos fazer uma página para exibir o livros de uma forma mais interessante para o administrador do sistema. Para deixar a nossa página de administração de produtos interessante, vamos utilizar um layout baseado no Bootstrap. Para isso, siga os seguintes passos:

   - no seu Desktop, clique na pasta que leva para os cursos da **Caelum**;
   - entra na pasta `JS-35/produtos`
   - copie o arquivo `lista.ejs` para a pasta `views/produtos` da sua aplicação
   - copie os arquivos da pasta `css` para a pasta `public/css` .
   - copie os arquivos da pasta `js` para a pasta `public/js` .

   Abra o arquivo **lista.ejs** e altere o conteúdo do arquivo para que possamos listar os livros. O ponto de alteração está marcado com um comentário HTML.

```
<h2 class="basic-title">Listagem de livros</h2>

<table class="table table-condensed table-bordered
    table-striped table-hover">
    <thead>
    <tr>
        <td>Titulo</td>
        <td>Preco</td>
        <td>Descricao</td>
    </tr>
    </thead>
    <tbody>
    <% for(var i=0; i<lista.length; i++) {%>
        <tr>
            <td><%= lista[i].titulo %></td>
            <td><%= lista[i].preco %></td>
            <td><p><%= lista[i].descricao%></p></td>
        </tr>
    <% } %>
    </tbody>
</table>
```

9. Como adicionamos arquivos estáticos a nossa aplicação, é necessário configurar o express para liberar o acesso a tais arquivos sem a necessidade de configuração de uma rota. Para fazer isso, vamos alterar o arquivo *custom-express.js* .

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

module.exports = function() {

    var app = express();

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

    ...
```

   A função *static* disponível no módulo do express faz uso da lib **serve- static** para possibilitar a entrega de arquivos estáticos. Ela é integrada com o próprio express.

10. Agora, é só fazer nosso controller exibir esta página com os dados dos livros. Abra o arquivo **routes/produtos.js** e modifique o callback do resultado da consulta

```
...
connection.query('select * from livros',
    function(err, result,fields) {
        res.render("produtos/lista",{lista:result});
    }
);
...
```

11. Acesse novamente `http://localhost:3000/produtos` e veja o resultado.

2.5 ISOLANDO A CRIAÇÃO DA CONEXÃO

Para conseguir executar qualquer consulta no nosso banco de dados, antes de tudo, precisamos sempre criar uma conexão. Este é o tipo de código que tende a ficar espalhado por todo o sistema, já que ainda vamos criar outras funcionalidades que dependem do acesso as tabelas. Nesse momento temos o seguinte código.

```
app.get("/produtos",function(req,res) {
    var mysql  = require('mysql');

    var connection = mysql.createConnection({
        host     : 'localhost',
        user     : 'root',
        password : '',
        database : 'casadocodigo'
    });

    connection.query('select * from livros',
        function(err, result,fields) {
            res.render("produtos/lista",{lista:result});
        }
    );

    connection.end();
};
```

Caso seja necessário configurar um timeout de conexão com o banco de dados, em quantos lugares vamos ter que alterar? E se quisermos configurar um pool de conexões? Essas são preocupações que devemos ter quando estamos construindo qualquer aplicação. Para amenizar esse problema, podemos criar uma função cuja única responsábilidade é justamente a de criar uma conexão. Vamos olhar como seria seu uso no nosso projeto.

```
var dbConnection = require('../infra/createDBConection');

app.get("/produtos",function(req,res) {
    var connection = dbConnection();
```

```
    connection.query('select * from livros',
        function(err, result,fields) {
            res.render("produtos/lista",{lista:result});
        }
    );

    connection.end();
});
```

A ideia é criarmos um módulo que exponha um função de criação de conexão com o banco de dados e, desse jeito, deixamos todo esse código isolado em um ponto só do nosso sistema. Agora vamos criar um arquivo para conter esse módulo, criaremos ele na pasta **infra**.

```
//arquivo createDBConection.js

var mysql  = require('mysql');

function createDBConection(){
    return mysql.createConnection({
        host     : 'localhost',
        user     : 'root',
        password : '',
        database : 'casadocodigo'
    });
}

module.exports = createDBConection;
```

Perceba que não fizemos nada que ainda não tenhamos estudado. Criamos uma função de abertura de conexão e atribuimos ela a variável `exports` , para que ela possa ser utilizada por todos os arquivos que carregarem este módulo.

Pronto, agora temos um código muito mais limpo e que atinge o mesmo resultado. No mundo de desenvolvimento orientado a objetos, o qual podemos fazer algum paralelo com o desenvolvimento em JavaScript, quando criamos uma função cujo único objetivo é construir um objeto um pouco mais complicado, dizemos que estamos seguindo um *Design Pattern* chamado de *Factory Method*. Esta é uma das várias soluções catalogadas no livro *Design Patterns*. É necessário lembrar que o JavaScript te permite seguir mais de um paradigma de desenvolvimento, mas a nossa solução lembra muito esse pattern, então é justo que façamos a associação.

2.6 EXERCÍCIOS: USANDO O FACTORY METHOD PARA ISOLAR A CRIAÇÃO DA CONEXÃO

1. Vamos criar um novo módulo que cuidará da criação da conexão com o banco. Crie o arquivo **connectionFactory.js** dentro da pasta **infra**:

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

function createConnection(){
    return mysql.createConnection({
```

```
        host: 'localhost',
        user: 'root',
        password: '',
        database: 'casadocodigo'
    });
};

module.exports = createConnection;
```

2. Abra o arquivo **routes/produtos.js** e troque a criação da conexão pelo módulo que acabamos de criar. Para acessar o módulo, receba o objeto `app` na função de criação do módulo:

```
var connectionFactory = require('../infra/connectionFactory');

module.exports = function(app){
    app.get("/produtos",function(req,res) {

        var connection = connectionFactory();

        connection.query('select * from livros',
            function(err, result,fields) {
                res.render("produtos/lista",{lista:result});
            }
        );

        connection.end();
    });
}
```

3. Reinicie o servidor e acesse `http://localhost:3000/produtos` e veja se tudo está funcionando ainda.

2.7 ISOLANDO A EXECUÇÃO DAS QUERIES E O DESIGN PATTERN DAO

Já discutimos um pouco que, idealmente, as funções devem ter responsabilidades únicas. Começamos a resolver esse problema no *Controller* isolando a criação de conexão e fazendo com que ele use a nossa *factory*. Um outro ponto que merece nossa atenção é em relação a execução das queries. Atualmente deixamos que o *Controller* passe o SQL necessário e, além disso, ele também é obrigado a lidar com argumentos específicos do banco. Por exemplo o argumento que representa a lista de colunas do banco não é nada útil na maioria dos casos. Indo um pouco mais além, o que realmente reresenta essa linha?

```
connection.query('select * from livros' ...)
```

Óbvio que como esse SQL é simples, qualquer um pode olhar e entender. E se a consulta fosse um pouco mais complicada?

```
connection.query('select from livros l inner join livros_autores la
on l.id=la.livor_id inner join autores a on a.id=la.autor_id
where l.preco > ... ')
```

Possivelmente o programador já ia demorar um pouco mais para entender. Muitas vezes devemos isolar um código não porque ele se repete e sim porque certas linhas de código merecem mais semântica.

É exatamente isso que vamos fazer agora no sistema, vamos isolar o código de consultas aos dados, para que possamos ter um código mais claro de ser entendido. O nosso objetivo é ter algo parecido com o que segue.

```
app.get("/produtos",function(req,res) {

    var connection = connectionFactory();
    var produtos = produtosNoBanco();

    produtos.lista(function(error,results,fields){
        res.render("produtos/lista",{lista:results});
    });

    })
    connection.end();
});
```

Perceba que agora o código da listagem do nosso *controller* ficou mais direto. O objetivo da função **produtos** é justamente a de retornar uma espécie de contexto onde sejamos capazes de invocar funções relativas a consultas de produtos no banco de dados. Por exemplo, criamos lá dentro uma função chamada *lista*, responsável pela listagem completa de produtos.

Perceba que o controller tem apenas a responsabilidade de passar o *callback* para que ele possa pegar o resultado e mandar para a página. O nosso único trabalho é criar mais um módulo na pasta de *infra*, vamos chamar esse novo arquivo de **produtos.js**.

```
//produtos.js

module.exports = function() {
        this.lista = function(connection,callback) {
            connection.query('select * from livros',callback);
        }
        return this;
}
```

Aqui usamos mais um conceito do JavaScript, que é a *função construtora*. A ideia é que agrupemos um conjunto de funções relacionadas a um mesmo contexto, que nesse caso é isolar as consultas de banco de dados relativos aos produtos. Esse é um conceito similar ao de **classe**, dentro da Orientação a Objetos. E aí, para cada nova representação dessa **classe** que for necessária na aplicação, criamos um novo **objeto**. Pensando um pouco mais a frente, já podemos imaginar que teremos algumas funções nessa classe.

```
module.exports = function() {
        this.lista = function(connection,callback) {
            connection.query('select * from livros',callback);
        }

        this.salva = function(connection,produto,callback) {
            ...
        }

        this.buscaPorNome = function(connection,nome,callback) {
            ...
```

```
        }
        return this;
}
```

Perceba que em todas as funções vamos precisar do parâmetro que representa a conexão. Pensando no design do código, o melhor é receber esse argumento apenas uma vez e poder usá-lo em todas funções que forem declaradas nesta classe. Para resolver isso podemos receber o argumento que representa a conexão na função construtora exportada pelo módulo.

```
module.exports = function(connection) {
        this.salva = function(livro,callback) {
            ...
        }

        this.lista = function(callback) {
            connection.query('select * from livros',callback);
        }

        this.buscaPorNome = function(nome,callback) {
            ...
        }
        return this;
}
```

Tudo que recebemos como argumento na função fica disponível para todas as outras funções declaradas dentro dela! Por exemplo, poderíamos ter o seguinte trecho de código.

```
var produtos = produtosNoBanco(connection);
produtos.buscaPorNome(nomeDoNovoProduto,function(result){
    if(result == null){
        produtos.salva(novoProduto);
    }
});
```

Invocamos funções no mesmo objeto duas vezes e, perceba, ambas utilizam a *connection* passada na função construtura. O nosso objeto está guardando estado! Na Orientação a Objetos dizemos que temos um **atributo**.

Essa solução de isolar o código de acesso a dados em um local da aplicação é um outro Design Pattern conhecido como **Data Access Object**. De novo, é um padrão aplicado a linguagens que suportam o conceito de objetos e, como o JavaScript implementa esses conceitos, podemos fazer uso do mesmo no nosso código. Na verdade, na última versão da linguagem, inclusive introduziram a nova palavra chave **class**. Justamente para deixar isso ainda mais claro.

Para deixar o uso do Design Pattern mais claro no nosso código, podemos inclusive alterar o nome do nosso arquivo para **produtoDao.js**.

2.8 USANDO O OPERADOR NEW

Acabamos de isolar os códigos do nosso sistema e até usamos o conceito de orientação a objetos. Só que o JavaScript é uma linguagem que muitas vezes nos permite escrever um mesmo pedaço de código de formas diferentes. Por exemplo, para criarmos nosso objeto podemos usar o seguinte código:

```
...
var produtos = produtosNoBanco(connection);
...
```

Esse exemplo até que funciona, mas vamos analisar um detalhe curioso. Caso peguemos o retorno da função construtora, responsável por criar nosso objeto, e fizermos a impressão dela no *console*, temos a seguinte saída.

```
{ DTRACE_NET_SERVER_CONNECTION: [Function],
  DTRACE_NET_STREAM_END: [Function],
  DTRACE_NET_SOCKET_READ: [Function],
  DTRACE_NET_SOCKET_WRITE: [Function],
  DTRACE_HTTP_SERVER_REQUEST: [Function],
  DTRACE_HTTP_SERVER_RESPONSE: [Function],
  DTRACE_HTTP_CLIENT_REQUEST: [Function],
  DTRACE_HTTP_CLIENT_RESPONSE: [Function],
  global: [Circular],
  process:
   { title: 'node',
     version: 'v0.12.4',
     moduleLoadList:
      [ 'Binding contextify',
        'Binding natives',
        'NativeModule events',
        'NativeModule util',
        'NativeModule buffer',
        'Binding buffer',
        'Binding smalloc',
        'NativeModule path',
        'NativeModule module',
        'NativeModule vm',
        'NativeModule assert',
        'NativeModule fs',
```

Perceba que o resultado é meio estranho, dado que só declaramos a função *lista* dentro da classe

deste objeto. O problema é justamente a maneira que invocamos a função construtora junto com o retorno dela.

```
module.exports = function() {
        this.lista = function (callback) {
            connection.query('select * from livros', callback);
        };

        return this;
}
```

A palavra reservada **this** representa justamente o objeto que está usando as funções e atributos da classe em determinado momento. A depender de como invocamos qualquer função, a referência que o **this** guarda pode mudar. Da maneira que estamos fazendo, o **this** está atrelado ao objeto global da nossa aplicação enquanto que, na verdade, nós queremos que ele represente apenas a a estrutura da nossa classe. Para ficar ainda mais claro, faremos uma brincadeira no nosso código. Suponha que a nossa classe defina mais um argumento na função construtora.

```
module.exports = function() {
    return function(connection,valorExtra) {
        this.valorExtra = valorExtra;

         ...
    }
}
```

E agora vamos testar um caso onde queremos criar dois objetos.

```
var produtoDao = ProdutoDao(connection,"teste");
var produtoDao2 = ProdutoDao(connection,"teste2");

console.log(produtoDao.valorExtra);

...
```

Pensando que o **this** sempre representa o objeto que faz a invocação, deveria ser impresso o valor **teste**, já que este foi o valor passado para a construção do primeiro objeto. Só que quando rodamos esse código o valor impresso é **teste2**. Esse exemplo só serve para corroborar que realmente estamos trabalhando apenas com um objeto, ao invés de vários. Para resolvermos esse problema, podemos fazer uso da palavra reservada **new**.

```
var produtoDao = new ProdutoDao(connection,"teste");
var produtoDao2 = new ProdutoDao(connection,"teste2");

console.log(produtoDao.valorExtra);


...
```

Pronto! Agora vai ser impresso o valor que esperamos. O objetivo do **new** é justamente de criar um novo contexto de uso para a sua classe, literalmente um novo objeto! Inclusive, se fizermos o mesmo teste de realizar a impressão da referência retornada pela função construtora, veremos a diferença.

```
{ lista: [Function]}
```

2.9 A PROPRIEDADE PROTOTYPE

O nosso código já ficou bastante modularizado. Criamos arquivos e classes separadas para separarmos as responsabilidades dos mesmos e atingimos um código bem legível. Só que agora temos mais um detalhe específico do JavaScript que vamos ter que lidar. Sempre que queremos criar um novo objeto do nosso *DAO* de produtos, invocamos o seguinte código.

```
function(connection) {
        this.lista = function (callback) {
            connection.query('select * from livros', callback);
        };

        return this;
}
```

Essa implementação funciona, só que para cada novo objeto que queremos criar o interpretador de JavaScript, no caso a *V8*, vai ter que interpretar o conteúdo da função e recriar toda estrutura na memória! Olhando com um pouco mais de cuidado o que muda de uma invocação da função *lista* para outra é o parâmetro e o atributo que guarda a conexão com o banco de dados, a estrutura em si nunca muda. Para resolver este tipo de cenário, o JavaScript permite que a gente defina a estrutura apenas uma vez e reaproveite a mesma para a criação de todos os objetos.

```
function ProdutoDao(connection) {
    this.connection = connection;
}

ProdutoDao.prototype.lista = function(callback) {
    this.connection.query('select * from livros',callback);
};

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

A primeira parte do trecho do código acima não muda muito em relação ao que a gente já viu. Criamos uma função construtora e guardamos a connection passada como atributo no objeto. A diferença total está na declaração das outras funções da classe. Para adicionar a função *lista* usamos a propriedade **prototype**, presente em todos os objetos criados no JavaScript.

Perceba que sempre quando criamos um novo objeto no JavaScript já ganhamos algumas funções que nem escrevemos. Elas existem justamente na representação básica de todo objeto em JavaScript, chamada de **Object**. Só temos acesso a essas funções porque elas são adicionadas no atributo **prototype** do **Object**.

Como queremos criar algumas funções que sejam compartilhadas entre todos os novos objetos que forem criados a partir da nossa função construtora, podemos adicioná-las justamente no **prototype**. Por fim, a nossa váriavel **module.exports** exporta uma função que retorna a referência para a nossa função construtora, cujo o nome é **ProdutoDao**.

Um detalhe importante aqui foi que tivemos de guardar o argumento da conexão no atributo chamado *connection*. Quando estávamos usando a outra estrutura, podíamos usar diretamente o argumento, já que todas as funções estavam no mesmo escopo. Em geral isso não é problema, mas para deixar claro que este atributo é só de uso interno, existe uma convenção de usar um _ na frente da variável que deve ser usada apenas de maneira privada.

```
function ProdutoDao(connection) {
    this._connection = connection;
}

ProdutoDao.prototype.lista = function(callback) {
    this._connection.query('select * from livros',callback);
};
```

Um último ponto é o escopo das funções criadas no nosso módulo. As únicas que ficam acessíveis de fora do módulo são as que são atribuidas a variável *exports*. Nesse caso, ninguém pode chamar diretamente a função **ProdutoDao**.

2.10 EXERCÍCIOS: ISOLANDO O CÓDIGO DE ACESSO A DADOS

1. Vamos isolar nossas consultas em um objeto separado. Crie o arquivo **ProdutoDao.js** dentro do diretório **infra** e adicione a consulta nele:

```
function ProdutoDao(connection) {
    this._connection = connection;
};

ProdutoDao.prototype.lista = function (callback) {
    this._connection.query('select * from livros', callback);
};

module.exports = ProdutoDao;
```

2. Importe o novo módulo em nosso controller. Dentro de **routes/produtos.js**:

```
var connectionFactory = require('../infra/connectionFactory');
var ProdutoDao = require('../infra/ProdutoDao');

...

    app.get("/produtos",function(req,res) {

        var connection = connectionFactory();
        var produtoDao = new ProdutoDao(connection);

        produtoDao.lista(function(error,results,fields){
            res.render("produtos/lista",{lista:results});
        });

        connection.end();
    });
...
```

3. Reinicie o servidor e acesse `http://localhost:3000/produtos` e veja se tudo está funcionando.

CAPÍTULO 3

COMPLETANDO O CADASTRO

Já estamos listando os livros que existem no banco de dados. Agora chegou a hora de possibilitarmos que eles sejam cadastrados pelos usuários. O interessante é que quase tudo que for necessário para implementar essa funcionalidade, já foi estudado no capítulo anterior. Este um momento de rever alguns dos conceitos!

3.1 RECEBENDO OS DADOS DO FORMULÁRIO

A primeira tarefa que devemos fazer é justamente criar uma página onde os dados do produto podem ser inseridos. Seguindo a convenção, vamos criar essa página na pasta específica para ela, *views/produtos/form.ejs*.

```
<html>
    <body>
        <form action="/produtos" method="post">
            <div>
                <label> Titulo do livro
                    <input type="text" name="titulo"
                        placeholder="nome do produto"/>
                </label>
            </div>
            <div>
                <label> Preço
                    <input type="text" name="preco"
                        placeholder="preço do produto"/>
                </label>
            </div>
            <div>
                <label> Descricao
                    <textarea name="descricao"></textarea>
                </label>
            </div>
            <input type="submit" value="gravar"/>
        </form>
    </body>
</html>
```

Lembre que os arquivos que estão na pasta *views* não são acessíveis diretamente por nenhum cliente. Por exemplo, caso alguém tente acessar o endereço http://localhost:3000/views/produtos/form.ejs, vai receber a seguinte mensagem do servidor: "Cannot GET /views/produtos/form.ejs".

Para que possamos acessá-la, precisamos criar uma nova rota dentro do *express*. Com estamos trabalhando com o módulo de produtos, podemos alterar o arquivo *routes/produtos.js*.

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

    ...

    app.get("/produtos/form",function(req, res) {
        res.render('produtos/form');
    });


}
```

Nada que não tenhamos feito! Agora vamos focar no código necessário para salvar um um novo livro.

```
app.get("/produtos",function(req,res) {
    var livro = req.body;
    console.log(livro);
    ...
});
```

Todas as informações que são enviadas em uma requisição estão disponíveis na variável *req*, que no caso representa a requisição em si. O importante para a gente agora é a propriedade *body*, que retorna os dados preenchidos no formulário de cadastro. Essa propriedade deveria retorna um JSON, que como já discutimos, é o formato de dados preferidos para trabalhar em JavaScript. Quando rodamos nossa aplicação o valor impresso é **undefined**. O problema é que o *express* espera que algum módulo seja instalado, alterando a estrutura do objeto que representa o *request*, justamente adicionando essa propriedade.

É justamente nesse cenário que entra a biblioteca **body-parser**. Como sempre, precisamos adicioná-la como um módulo da aplicação.

```
npm install body-parser --save
```

Agora precisamos ensinar ao *express* que ele deve usar o **body-parser** para recuperar os parâmetros enviados na requisição e deixar disponível na propriedade **body**. Como isso é um código de configuração, vamos deixá-lo no arquivo *custom-express.js*. O código deve vir antes do carregamento das rotas.

```
var express = require('express');
var bodyParser = require('body-parser');

module.exports = function() {

    ...
    app.use(bodyParser.urlencoded());

    require('./routes/produtos')(app);
    return app;
};
```

Pronto! Agora, se voltarmos a tentar cadastar um novo livro a partir do formulário criado, o `console.log(req.body)` deve exibir as informações enviados através do JSON retornado.

```
{ titulo: 'livro de node',
```

```
  preco: '59.90',
  descricao: 'Básico sobre nodejs'
}
```

Com os dados na nossa mão, só precisamos agora criar o resto do código que possibilita a inserção no banco. Nesse momento, tudo vai ser muito parecido com o que fizemos na listagem.

```
function(req,res) {
    var livro = req.body;

    var connection = app.infra.connectionFactory();

    var produtos = new app.infra.ProdutoDao(connection);

    produtos.salva(livro,function(exception,result){
        if(!exception) {
            res.render("produtos/salvo");
        } else {
            //exemplo jogando para uma página de erro
            res.render("produtos/erro");
        }
    });
};
```

A única diferença do código acima para a listagem, é que invocamos a função *salva* ao invés da *lista*. Caso nenhum problema tenha ocorrido, direcionamos o usuário para uma tela, informando que deu tudo certo. Para não termos um código mágico, vamos dar uma olhada na função que salva um livro. Para manter a coesão do sistema, ela foi criado dentro do *DAO*.

```
function ProdutoDao(connection) {
    this._connection = connection;
}

ProdutoDao.prototype.salva = function(livro,callback) {
    this._connection.query('insert into livros SET ?', livro, callback);
}

ProdutoDao.prototype.lista = lista = function(callback) {
    this._connection.query('select * from livros',callback);
}

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

Perceba que usamos a mesma função *query*, só que agora passamos um **insert**, ao invés de um **select***. O outro detalhe interessante é que nem foi necessário concatenar os valores. Só marcamos com a **?** que ali vai entrar um parâmetro, que no caso é o JSON com as informações do novo livro. Como o JSON é uma estrutura baseada em chaves e valores, o próprio *driver* já é capaz de gerar o resto do comando SQL necessário. Veja o que foi gerado logo abaixo.

```
INSERT INTO livros SET `titulo` = 'livro de node',
    `preco` = 59.9, `descricao` = 'Básico sobre nodejs'
```

Mapeando a rota para cadastro

O nosso formulário aponta para a URL **/produtos**. E agora, na hora que fomos mapear, como vai ficar?

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

    app.get("/produtos/form",...);
    //rota para salvar
    app.get("/produtos",...);
    //rota para listar
    app.get("/produtos",...);
}
```

O problema aqui é que estamos deixando de usar um recurso importante do protocolo HTTP, os verbos.

- `POST` : usado quando existe a necessidade de criação de algum recurso;
- `GET` : usado quando o interesse é o de recuperar alguma informação;
- `DELETE` : como o nome diz, deve ser usado para excluir algum recurso;
- `PUT` : associado com alguma operação de atualização de recursos no servidor.

Estes talvez sejam os verbos mais conhecidos, com destaque para o `post` e o `get` . Perceba que a função que invocamos já informa o verbo que será aceito para a requisição.

O verbo, junto com a URL, serve justamente para indicar o tipo de operação que deve ser realizada no servidor. Por exemplo, se o endereço `/produtos` for acessado através de um `get` , quer dizer que uma listagem está sendo requisitada. Caso seja acessada através de um `post` , está sendo pedido para um novo livro ser criado. Vamos ver como fica a representação dessa teoria em nosso código:

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

    app.get("/produtos/form",...);
    //post para salvar um produto
    app.post("/produtos",...);
    //get para listar os produtos
    app.get("/produtos",controller.lista);
}
```

3.2 EXERCÍCIOS: CADASTRANDO UM NOVO LIVRO

1. Para fazermos nosso cadastro, vamos seguir o mesmo fluxo realizado na listagem. Importaremos o arquvivo já com o html base todo pronto. Siga os seguintes passos:

   - no seu Desktop, clique na pasta que leva para os cursos da **Caelum**;
   - entra na pasta `JS-35/produtos`
   - copie o arquivo `form.ejs` para a pasta `views/produtos` da sua aplicação

   Abra o arquivo **form.ejs** e altere o conteúdo do arquivo para que possamos adicionar o formulário de criação. O ponto de alteração está marcado com um comentário HTML.

```
<form action="/produtos" method="post" class="well" role="form">
    <div class="form-group">
        <label for="titulo">Titulo</label>
        <div class="input-group">
            <span class="input-group-addon">
                <i class="glyphicon glyphicon-unchecked"></i>
            </span>
            <input type="text" name="titulo"
                placeholder="titulo do livro"/>
        </div>
    </div>
    <div class="form-group">
        <label for="preco">Preço</label>
        <div class="input-group">
            <span class="input-group-addon">
            <i class="glyphicon glyphicon-unchecked"></i>
            </span>
            <input type="text" name="preco"
            placeholder="valor do livro"/>
        </div>
    </div>
    <div class="form-group">
        <label for="descricao">Descricao</label>
        <div class="input-group">
            <span class="input-group-addon">
```

```
            <i class="glyphicon glyphicon-unchecked"></i></span>
            <textarea name="descricao"></textarea>
        </div>
    </div>
    <button type="submit" class="btn btn-primary">Gravar</button>
</form>
```

2. Crie a rota para acessar o formulário. Abra o arquivo **routes/produtos.js** e adicione a rota:

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

    app.get("/produtos/form", function(req,res){
        res.render('produtos/form');
    });
};
```

3. Reinicie o servidor e acesse **http://localhost:3000/produtos/form**. Veja se o formulário foi exibido corretamente.

4. Para poder receber os dados do livro no formulário, vamos instalar o **body-parser**:

```
npm install body-parser --save
```

5. Configure o `express` para usar o `body-parser` no arquivo **custom-express.js` :

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

module.exports = function() {
    ...
    app.use(bodyParser.urlencoded());

    require('./routes/produtos')(app);
    return app;
};
```

6. Registre no arquivo de rotas o endereço para submeter o formulário. Abra **routes/produtos.js** e insira a nova rota associada a função que trata a mesma.

```
    ...
    app.post("/produtos",function (req, res) {
        var livro = req.body;

        var connection = app.infra.createConnection();
        var produtos = new app.infra.ProdutoDao(connection);

        produtos.salva(livro, function (exception, result) {
                res.render("produtos/salvo");
        });
    });
    ...
```

7. Ao salvar um produto com sucesso, vamos exibir uma página com uma mensagem. Crie um novo arquivo em **views/produtos/salvo.ejs** :

```
<html>
```

```
<body>
    Produto salvo com sucesso!
</body>
</html>
```

8. Antes de testar ainda precisamos implementar o método que faz o insert no banco. Abra o arquivo **infra/ProdutoDao.js** e adicione o método que salva um produto:

```
...
ProdutoDao.prototype.salva = function(livro,callback) {
    this._connection.query('insert into livros SET ?', livro, callback);
};
...
```

9. Reinicie o servidor e tente cadastrar um novo produto. Para conferir se o livro foi inserido corretamente, consulte o banco de dados ou acesse a página da listagem de produtos.

3.3 UM POUCO MAIS SOBRE O PROTOCOLO HTTP

Agora que já inserimos e listamos, chegou a hora de melhorar um pouco fluxo entre essas operações. Neste momento, quando um produto é inserido, o usuário é levado para uma página chamada `salvo.ejs`, que apenas indica que um novo livro foi cadastrado com sucesso. Nesse tipo de cenário, o fluxo mais indicado é voltar com o usuário para a listagem, talvez mostrando uma mensagem de sucesso.

```
var produtos = new app.infra.ProdutoDao(connection);

produtos.salva(livro,function(exception,result){
    if(!exception) {
        controller.lista(req,res);
    }
});
```

Esse código implementa justamente o fluxo sugerido. Quando acabamos de salvar um novo produto, invocamos o método `lista`, que é o responsável por listar os produtos e jogá-los para a página `produtos/lista.ejs`. O ponto negativo dessa solução é que o endereço que fica na barra do navegador ainda é o último acessado pelo usuário, que nesse caso foi um `post` para `/produtos`. Caso o nosso cliente aperte um `F5`, o navegador vai tentar refazer a última operação, causando uma nova inserção de produto no sistema.

É considerada uma má prática realizar um *render* após o usuário ter feito um `post`, justamente por conta do problema da atualização. Para esse cenário, a melhor solução é forçar o usuário a fazer uma nova requisição para a nossa listagem e, dessa forma, permitir que ele atualize a página sem que um novo `post` seja realizado.

```
var produtos = new app.infra.ProdutoDao(connection);

produtos.salva(livro,function(exception,result){
    if(!exception) {
        res.redirect("/produtos");
    }
});
```

A função `redirect:` indica para o *express* que, em vez de simplesmente fazer um forward, é necessário que ele retorne o status 302 para o navegador, solicitando que o mesmo faça um novo request para o novo endereço.

Entre cabeçalhos contidos na resposta existe um chamado **Location**, que informa justamente qual é o endereço ao qual o navegador deve fazer a próxima requisição. Essa técnica, onde fazemos um redirect do lado do cliente logo após um `post`, é um padrão conhecido da web chamado de **Always Redirect After Post** e deve ser sempre utilizado, principalmente quando o cliente é uma pessoa usando um navegador.

3.4 EXERCÍCIOS: REDIRECT AFTER POST

1. Vamos alterar o controller para após salvar um novo produto redirecionar para a lista ao invés da página com a mensagem. Abra o arquivo **routes/produtos.js** e altere o método `salva%% para fazer o redirecionamento ao invés de abrir a página:

```
...
produtos.salva(livro, function (exception, result) {
    res.redirect("/produtos");
});
...
```

2. Reinicie o servidor e cadastre um novo produto. Veja será redirecionado para a listagem após a inserção.

POR QUE ESTAMOS USANDO CALLBACKS O TEMPO TODO?

Uma pergunta que pode passar pela nossa cabeça é: por que o tempo todo estamos passando funções de callbacks quando queremos fazer alguma coisa? Só para refrescar a memória, vamos ver um código que escrevemos para gravar um livro no banco de dados.

```
produtos.salva(livro, function (exception, result) {
    res.redirect("/produtos");
});
```

Invocamos o método `salva` passando como argumento um livro e uma função que deve ser chamada quando o acesso ao banco de dados tiver acabado. No mundo normal, esse mesmo código seria da seguinte maneira:

```
produtos.salva(livro);
res.redirect("/produtos");
```

Invocamos um método e a linha de baixo só é executada quando a de cima acaba, é o típico código que chamamos de **síncrono**. O problema desse código, considerando um servidor web convencional, é que enquanto estamos fazendo o acesso ao banco, uma operação de **I/O**, o servidor fica parado sem

poder atender nenhum nova requisição vinda de um cliente, por exemplo um navegador.

Para resolver este tipo de situação, os servidores criam novas **Threads** para que possam atender o máximo de requisições possíveis. Por mais que essa solução funcione bem, afinal de contas ela vem sendo usada por muito tempo por servidores famosos em outras linguagens, como o Tomcat no JAVA, estamos no fim desperdiçando recursos.

Quantas requisições poderiam ser atendidas enquanto uma Thread está parada esperando o retorno do banco? Ou de alguma integração com outro serviço externo via Web Service? É justamente aí que o Node.js entra! Todas as operações dentro do Node são feitas de maneira assíncrona. E uma parte muito importante, as bibliotecas construídas em cima da plataforma dele compraram a ideia, então todas funcionam em cima do modelo assíncrono imposto por ele.

Quando usamos o driver do MYSQL e invocamos um método que conecta no banco, essa operação é feita de maneira não blocante e, quando ela estiver pronta nosso callback é chamado. Nesse meio tempo o servidor pode ir atendendo outras requisições. Dessa maneira, a mesma Thread rodando no Node.js consegue atender um número maior de requisições, deixando a aplicação mais fácil de escalar em cenários de muitos requests. A diferença é tão absurda que, por padrão, o Node.js funciona apenas com **uma** thread! Caso queiramos tirar proveito de todos os núcleos de processamento do servidor, somos obrigados a subir outras instâncias da nossa aplicação.

O preço que a gente paga é ter que usar callbacks o tempo todo, o que torna o modelo de programação não tão natural.

CAPÍTULO 4

RESPONDENDO MAIS DE UM FORMATO

Nossa aplicação, até o presente momento, só lida com requisições vindas de um navegador, só que agora vamos um pouco além do que já existe na Casa do Código. Sites de vendas muito grandes, como a Amazon e o Submarino, além de exporem os seus produtos em seus sites, fazem parcerias com outros sites para que essas outras aplicações também possam exibir seus produtos.

4.1 EXPONDO OS DADOS EM OUTROS FORMATOS

Quando falamos de integração com outras aplicações, o primeiro ponto em que temos que pensar é: qual é o formato que vamos usar para realizar a integração? Atualmente, nossa aplicação só é capaz de retornar páginas para os clientes, no caso os navegadores, em `HTML`. Um outro tipo de cliente, hoje já muito comum, são os celulares com Android ou iOS. E como você já deve esperar, exibir dados através de `HTML` pode não ser o formato ideal para ser usado nesses aparelhos. Pensando no próprio ecossistema da Casa do Código, outras aplicações podem necessitar de integrações para processar informações dos livros.

A primeira tarefa que precisamos fazer é mapear outra rota, que seja capaz de retornar a lista de livros em outro formato. O escolhido aqui vai ser o `JSON`, que é um formato muito comum no mercado e muito fácil de ser gerado por uma aplicação escrita em JavaScript.

```
app.get("lista/json",function(req,res) {

    var connection = app.infra.connectionFactory();

    var produtos = new app.infra.ProdutoDao(connection);

    produtos.lista(function(error,results,fields){
        res.json(results);
    })
    connection.end();
});
```

A única diferença foi que usamos a função *json*, passando diretamente o retorno da nossa consulta. Ela informa para o *express* que ele deve escrever o resultado diretamente no *response*, já informando para o cliente que o tipo de resposta é JSON. Isso é feito através do cabeçalho *Content-type*, que já existe no HTTP e é entendido por qualquer cliente que suporte o protocolo. Também poderíamos ter usado a função *send*. Ela é um pouco mais esperta e configura o *Content-type* baseado no tipo do argumento.

Agora basta que alguém acesse o endereço `/produtos/json`, que o retorno já vai ser um `JSON`.

Nesse momento, temos que ter um método para cada formato de resposta que precisamos. Perceba que o código dos dois, muitas vezes, vai ser o mesmo.

4.2 CONTENT NEGOTIATION

Ter um método para cada formato de resposta diferente até funciona, o único problema é que você vai acabar com códigos repetidos. Lembre-se que o único ponto que vai mudar é a representação do retorno, a lógica para recuperar o dado vai ser a mesma. Vamos tentar resolver este problema. Primeiro, precisamos apagar a nova função, a nova rota e deixar apenas o que já existia, que já adiciona a lista de produtos numa página.

```
app.get("/produtos",function(req,res) {

    var connection = app.infra.connectionFactory();

    var produtos = new app.infra.ProdutoDao(connection);
    produtos.lista(function(error,results,fields){
        res.render("produtos/index",{lista:results});
    });

    })
    connection.end();
});
```

O ponto agora é: quando um cliente fizer o request para a URL `/produtos` , qual formato vamos retornar? Essa é a parte interessante, o protocolo HTTP já fornece um jeito de lidar com esse problema! Ele permite que o cliente indique qual o formato de resposta que ele prefere. Segue um exemplo de requisição usando a ferramenta **CURL**, famosa no mundo Unix.

```
curl -H "Accept: application/json" http://localhost:3000/produtos

curl -H "Accept: text/html" http://localhost:3000/produtos
```

No primeiro exemplo, fizemos uma requisição indicando que desejamos o retorno no formato

`application/json`. Isso é feito através do cabeçalho `Accept`. Perceba que no segundo já indicamos que queremos `HTML` como formato. Essa técnica é conhecida como **Content Negotiation** e é muito utilizada em integrações baseadas no HTTP, também conhecida como REST.

A parte interessante é que o *express* já oferece esse suporte para nós. Precisamos apenas ensiná-lo que agora ele tem que decidir qual formato retornar baseado no `Accept`.

```
app.get("/produtos",function(req,res) {

    var connection = app.infra.connectionFactory();

    var produtos = new app.infra.ProdutoDao(connection);
    produtos.lista(function(error,results,fields){
        res.format({
            html: function(){
                res.render("produtos/lista",{lista:results});
            },
            json: function(){
                res.json(results);
            }
        });
    });

    })
    connection.end();
});
```

A função *format* recebe um JSON com chaves já definidas pelo *express* e, baseado no **Accept**, ele decide qual função usar. Dessa forma, caso a lógica realmente seja a mesma, você pode aproveitar a mesma função.

4.3 EXERCÍCIOS: SUPORTANDO CONTENT NEGOTIATION

1. Vamos alterar nosso método que lista os produtos para suportar vários formatos. Abra o arquivo `routes/produtos.js` e modifique o controller associado a rota `/produtos`.

```
app.get("/produtos",function(req,res) {

    var connection = app.infra.connectionFactory();

    var produtos = new app.infra.ProdutoDao(connection);
    produtos.lista(function(error,results,fields){
        res.format({
            html: function(){
                res.render("produtos/lista",{lista:results});
            },
            json: function(){
                res.json(results);
            }
        });
    });

    })
    connection.end();
});
```

2. Reinicie o servidor. Acesse pelo navegador o endereço `http://localhost:3000/produtos` . A lista deve continuar sendo exibida corretamente em uma tabela html.

3. Para testar a renderização em *json* vamos usar o `curl` , uma ferramenta do terminal para fazer requisições http. Abra um terminal e execute:

```
curl -H "Accept: application/json" http://localhost:3000/produtos
```

   Veja se o retorno está no formato json.

4. (opcional) Teste pelo terminal o formato html também:

```
curl -H "Accept: text/html" http://localhost:3000/produtos
```

Ainda pensando em cenários de integração podemos, além de precisar listar os livros em formatos diferentes, cadastrar livros cujo os dados não vem necessariamente através de um formulário html. Quando preenchemos um formulário, numa página web, o navegador faz uma requisição informando que o formato de envio dados é o **x-www-form-urlencoded**. Essa informação é definida no *header Content-type*.

O problema é que temos um outro sistema na Casa do Código, que carrega um conjunto de novos livros definidos em um arquivo de texto, que também precisa realizar o cadastro na loja. Neste caso, um formato melhor de envio pode ser o próprio JSON. Podemos fazer a simulação usando o próprio *curl*.

```
curl -H "Content-type: application/json"
    -X POST
    -d '{"titulo":"novo livro","preco":150,
    "descricao":"descricao do livro"}'
    http://localhost:3000/produtos
```

Só que quando realizamos o *post* enviando os dados como JSON, é gravado tudo nulo no banco! Na verdade isso é até bom, imagine se nossa aplicação já aceitasse o JSON sem a gente tem feito nada, ia parecer mágica. O nosso problema é que não ensinamos ao módulo **body-parser** que ele também deve preencher a propriedade **body** do request, mesmo que o formato de envio seja JSON.

```
app.use(bodyParser.urlencoded());
```

Para também suportarmos JSON, é necessário invocar a função *json*, presente no **body-parser**.

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

Pronto! Agora nossa aplicação já suporta dois formatos de envio de dados e o melhor, deixamos a cargo do cliente para que ele decida qual é a melhor estratégia para ele. Uma das principais características do estilo arquitetural **REST** é que ele permite que as nossas integrações sejam flexíveis.

4.4 EXERCÍCIOS: SUPORTANDO JSON COMO FORMATO PARA CADASTRO

1. Abra o arquivo **custom-express.js** e configure o `body-parser` para suportar entrada de dados em json:

```
...
app.use(bodyParser.urlencoded());
app.use(bodyParser.json());
...
```

2. Reinicie o servidor. Teste a nova funcionalidade usando o `curl` no terminal (Digite tudo na mesma linha):

```
curl -X POST
   -H "Content-Type:application/json"
   -d
'{
    "titulo":"novo",
    "preco":150,
    "descricao":"descricao do livro"
}'
 'http://localhost:3000/produtos'
```

CAPÍTULO 5

VALIDANDO E OUTROS STATUS DO HTTP

O nosso cadastro já está funcional, mas estamos deixando de fazer uma coisa básica: validar os dados de entrada. Neste momento, caso uma aplicação ou um usuário queira cadastrar um novo livro, ele tem a opção de deixar todos os campos em branco.

5.1 VALIDANDO COM O EXPRESS-VALIDATOR

Para facilitar nossa vida, ao invés de ficar gastando tempo implementando código simples de validação, vamos usar um outro módulo construído para o *express*, chamado **express-validator**. Como é de praxe, a primeira coisa que precisamos fazer é instalar o módulo na aplicação.

```
npm install express-validator --save
```

Também precisamos carregá-lo, dentro da nossa aplicação. Então vamos alterar o arquivo *custom-express.js*.

```
var express = require('express');
var bodyParser = require('body-parser');
var expressValidator = require('express-validator')

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

    ...
    app.use(expressValidator());

    require('./routes/produtos')(app);

    return app;
};
```

Com tudo configurado, chegou a hora de efetivamente validarmos os dados que estão enviados para a nossa aplicação. Vamos ver como ficaria a validação do preço e do titulo do livro.

```
function(req,res) {
    var livro = req.body;

    req.assert('titulo', 'Titulo deve ser preenchido').notEmpty();
    req.assert('preco','Preco deve ser um número').isFloat();

    var errors = req.validationErrors();
```

```
    ...
}
```

O *express-validator* adiciona no request algumas novas funções relativas ao processo de validação. Nesse caso aqui estamos usando a *assert* e a função *errors*. A primeira retorna um objeto cujo o tipo é **ValidatorChain**. Este objeto possui as funções de validação que desejamos aplicar na propriedade especificada. No nosso caso, queremos saber se o titulo está preenchido e se o preço realmente é um número. Por debaixo do pano, o *express-validator* usa um outro módulo chamado *validator*, que fornece muitas funções de validação. Basicamente ele é a cola entre esse módulo e o *express*. A função *errors* é adicionada no request para possibilitar que recuperemos os possíveis erros gerados.

Caso algo tenha ocorrido, precisamos dar o feedback para o cliente.

```
var errors = req.validationErrors();
if(errors){
    res.format({
       html: function(){
           res.status(400).render("produtos/form",
               {validationErrors:errors});
       },
       json: function(){
           res.status(400).send(errors);
       }
    });
    return ;
}
```

Perceba que como nossa aplicação lida com mais de um formato de entrada de dados, somos obrigados a lidar com isso na hora de retornar as falhas de validação. O objeto *errors* é um JSON que contém as falhas de validação.

```
[
    {
        "param": "titulo",
        "msg": "Titulo deve ser preenchido",
        "value": ""
    },
    {
        "param": "preco",
        "msg": "Preco deve ser um número"
    }
]
```

Outros status do HTTP

O protocolo HTTP trabalha com diversos "status codes". Abaixo seguem os códigos mais conhecidos.

- 404; recurso não encontrado
- 500; problema no servidor;
- 200; tudo ocorreu da forma esperada;

O interessante é que temos muitas outras possibilidades. Por exemplo, quando nossa validação falha usamos o 400 (Bad Request).

```
res.status(400).send(errors);
```

Esse código indica que a requisição veio com dados inválidos. E essa é uma parte muito importante numa integração via REST. O cliente da aplicação vai se apoiar justamente nesse status para saber o que fazer com a resposta do servidor. Por exemplo, caso você retorne 200, ele pode achar que deu tudo certo na requisição, mesmo que o servidor tenha enviado um JSON com os erros.

Uma outra situação que usamos outro status foi no sucesso do cadastro de um produto.

```
produtoDao.salva(livro,function(exception,result){
    if(!exception) {
        res.redirect("/produtos");
    }
});
```

Quando usamos a função *redirect* estamos, na verdade, enviado o código 302 para o cliente. Ele indica que queremos que o cliente acesse uma nova URL. Dê uma olhada nas informações que são escritas no cabeçalho da resposta, quando usamos a função *redirect*.

```
HTTP/1.1 302 Moved Temporarily
...
Content-Type:text/html; charset=utf-8
Location:/produtos
```

Perceba que tem uma chave chamada de *location*. Pensando que o cliente é uma outra aplicação, ela agora sabe para onde ir. É justamente isso que o navegador faz!

5.2 EXERCÍCIOS: VALIDANDO E USANDO OS STATUS DO HTTP

1. Abra a pasta da aplicação no terminal e instale o `express-validator`:

```
npm install express-validator --save
```

2. Configure o validator no arquivo `custom-express.js`:

```
var express = require('express');
...
var expressValidator = require('express-validator')

module.exports = function () {
    ...
    app.use(expressValidator());
    ...
    return app;
};
```

3. Adicione as regras de validação no controller. Altere o arquivo **routes/produtos.js** :

```
...
app.post("/produtos",function (req, res) {
    var livro = req.body;

    req.assert('titulo', 'Titulo deve ser preenchido').notEmpty();
    req.assert('preco','Preco deve ser um número').isFloat();
    var errors = req.validationErrors();

    ...
});
```

4. Ainda no controller, precisamos testar se ocorreu algum erro de validação, e redirecionar uma resposta para o usuário:

```
...
var errors = req.validationErrors();

if (errors) {
    console.log("há erros de validação!");
    res.format({
        html: function () {
            res.status(400).render("produtos/form");
        },
        json: function () {
            res.status(400).send(errors);
        }
    });
    return;
}
...
```

5. Reinicie o servidor e tente cadastrar um produto sem informar o título. Veja se a mensagem de erro foi exibida no servidor e se o produto não foi cadastrado (consulte a lista)

6. Apesar de não estar cadastrando produtos sem dados no banco, ainda não estamos exibindo para o usuário nenhuma mensagem informando porque o produto não foi cadastrado. Vamos mudar isso. Primeiro, vamos disponibilizar para a view os erros de validação, usando o próprio método `render` do express:

```
...
html: function () {
    res.status(400).render("produtos/form",
                        { validationErrors: errors });
},
...
```

7. Abra a página em `views/produtos/form.ejs` . Agora que temos acesso aos erros, vamos exibí-los em uma lista:

```
<html>
<body>

<% if(locals.validationErrors){ %>
    <ul>
    <% for(var i=0; i< validationErrors.length; i++) {%>
        <li> <%= validationErrors[i].msg %>
    <% } %>
    </ul>
<% } %>

...
```

Da primeira vez que acessamos o formulário de cadastro ainda não existe nenhuma variável `validationErrors` . Caso ela fosse usada diretamente, receberíamos um erro de variável não encontrada. Acessando a mesma através do `locals` , simplesmente vamos receber um `undefined` e o `if` já encara isso como `false` .

5.3 EXERCÍCIOS OPCIONAIS: IMPLEMENTE A BUSCA DE PRODUTO POR ID

1. Implemente a busca por id. Ao acessar uma URL como `http://localhost:3000/produtos/1` deve retornar o formulário com os valores preenchidos, ou o json equivalente do produto de id 1. Para montar a rota, pesquise sobre **Express Route Parameters**.

Capítulo 6

TESTANDO SUA APLICAÇÃO

Nosso sistema já está com algumas funcionalidades implementadas. Gravamos, editamos e listamos os livros. Podemos ainda adicionar mais coisas, como só listar os livros que foram aprovados, colocar livros em destaque, associá-los às respectivas categorias etc. Você já é capaz de implementar tudo isso. Um detalhe muito importante, que não foi tratado até este momento, é a parte de testes da nossa aplicação. Não queremos ficar rodando tudo manualmente para saber se as coisas estão funcionando.

6.1 AUTOMATIZANDO A EXECUÇÃO DOS TESTES COM O MOCHA

A solução para não termos que ficar rodando tudo manualmente é escrever trechos de código que façam isso para a gente. No caso do JavaScript, a ideia é escrever algumas funções que já verifiquem os comportamentos de algum pedaço do nosso sistema. Para facilitar a escrita e execução destes testes, vamos utilizar uma biblioteca chamada **Mocha**.

Como já é de praxe, a nossa primeira tarefa é instalar o **Mocha** no nosso projeto.

```
npm install mocha --save-dev
```

!--@note https://docs.npmjs.com/cli/install --

Perceba que usamos uma opção a mais, a **-dev**. As *libs* usam essa opção para deixar claro que tais dependências não devem ser instaladas em ambiente de produção.

Agora que está tudo pronto, podemos começar a escrever nossos testes. O **Mocha** pede que seja criado uma pasta chamada *test*, ondem devem estar todos nossos arquivos de teste. Vamos dar uma olhada em um primeiro exemplo.

```
//produtos.js

describe('#ProdutosController', function() {

    it('#listagem de produtos json', function () {
        //o codigo do teste vai aqui
    });

});
```

Aqui já começamos a usar algumas funções do **Mocha**. A função *describe* é usado para darmos um contexto ao nosso teste. Perceba que estamos informando que queremos realizar testes sobre o controller de produtos. Agora, para cada cenário de teste que temos, invocamos a função *it*. A ideia é justamente

agrupar os contextos para que fique mais fácil de manter a nossa bateria de testes.

Um detalhe importante é que tanto a função *describe* quanto a *it*, recebem funções como argumentos. Tudo porque o **Mocha** mantém o estilo assíncrono suportado pelo Node.js! Para rodar os nossos testes, basta usarmos o executável que já vem dentro da biblioteca.

```
node_modules/mocha/bin/mocha test
```

Como instalamos ele dentro da nossa aplicação, somos obrigados a navegar pela pasta **node_modules**. A vantagem dessa abordagem é que garantimos que todos os integrantes do projeto vão usar a mesma versão do **Mocha**. A saída do nosso teste é algo parecido com o que segue.

```
#ProdutosController
 #listagem de produtos json


  1 passing (12ms)
```

O problema é que nosso teste ainda não está fazendo nada. Perceba que nesse primeiro exemplo, a nossa ideia é verificar se estamos respeitando o tipo de dados pedido pelo cliente. Precisamos ter certeza que o tipo de dados retornado pelo servidor foi exatamente o que foi solicitado pela aplicação cliente. Vamos tentar dar uma olha como seria esse código.

```
var http = require('http');
var assert = require('assert');

describe('#ProdutosController', function() {

    it('#listagem de produtos json', function () {

        var options = {
          hostname: 'localhost',
          port:3000,
          path: '/produtos',
          headers: {
            'Accept': 'application/json',
          }
        };

        http.get(options, function(res) {
          console.log(res.statusCode);
          console.log(res.headers['content-type']);
        });
    });

});
```

Para checar a listagem, é necessário que façamos uma requisição para a nossa aplicação informando que queremos um JSON como resposta. Perceba que usamos API do próprio Node.js para fazer o trabalho! Rodando o teste agora, teremos a seguinte saída:

```
#ProdutosController
    #listagem de produtos json
```

```
  1 passing (27ms)
```

Perceba que não mudou nada nossa saída. Alteramos bastante o corpo da função passada como argumento para o *it*, mas mesmo assim ainda não conseguimos nenhum feedback. Temos sempre que lembrar que qualquer trabalho de I/O que é realizado dentro do Node.js, é feito de maneira assíncrona. O que acontece é que tentamos realizar um *get* para o nosso endereço, só que, como essa requisição é feita de maneira não blocante, o **Mocha** não sabe tem que esperar a requisição voltar.

Já pensando neste tipo de situação, o **Mocha** permite que recebamos uma função como argumento que deve ser chamada exatamente no momento que queremos sinalizar que realmente nosso teste terminou.

```
it('#listagem de produtos json', function (funcaoDeFinalizacao) {

    var options = {
      hostname: 'localhost',
      port:3000,
      path: '/produtos',
      headers: {
        'Accept': 'application/json',
      }
    };

    http.get(options, function(res) {
      if(res.statusCode == 200){
        console.log("Status correto");
      } else{
        console.log("Status incorreto");
      }

      if(res.headers['content-type'] == 'application/json; charset=utf-8'){
        console.log("Formato retornado correto");
      } else {
        console.log("Formato retornado errado");
      }
      funcaoDeFinalizacao();
    });
});
```

Perceba que a *funcaoDeFinalizacao* só é invocada depois dos nossos `console.log` . Agora o **Mocha** sabe que deve travar a execução até que esta função seja invocada. Esse é um parâmetro muito importante, já que boa parte do tempo você vai lidar com códigos assíncronos. Vamos ver agora qual a saída do console.

```
#ProdutosController
Status correto
Formato retornado correto
    #listagem de produtos json (122ms)


  1 passing (135ms)
```

Asserts

Por mais que nosso teste já esteja rodando, ainda está um pouco complicado acompanhar o

resultado pelo terminal. O uso do `console.log` pode parecer que ajuda, mas imagine se estivéssemos rodando diversos testes, como saber o que deu certo e o que deu errado? Nesse momento teríamos que ficar analisando cada impressão. E quantos *ifs* serão necessários para cobrirmos todos os casos? Para facilitar um pouco o nosso trabalho, podemos usar o módulo de **asserções** fornecido pelo **Node.js**. Vamos dar uma olhada para ver como ficaria o código.

```
var http = require('http');
var assert = require('assert');

var options = {
  hostname: 'localhost',
  port:3000,
  path: '/produtos',
  headers: {
    'Accept': 'application/json',
  }
};

http.get(options, function(res) {
  assert.equal(res.statusCode,200);
  assert.equal(res.headers['content-type'],
  "application/json; charset=utf-8");

  done();
});
```

Perceba que carregamos o módulo chamado **assert**. Ele nos retorna um objeto que já possui umas funções que realizam justamente aqueles nossos *ifs*. Só que elas vão além e nos dão um feedback bom, quando alguma verificação falha. Vamos supor que esperamos o *statusCode* igual a 302.

```
#ProdutosController
    1) #listagem de produtos json


  0 passing (149ms)
  1 failing

  1) #ProdutosController #listagem de produtos json:

      Uncaught AssertionError: 200 == 302
      + expected - actual

      -200
      +302
```

Ele nos indica exatamente qual foi a asserção que falhou, facilitando muito a análise do resultado. O interessante é que basta que qualquer método lance uma exception do tipo **AssertionError** para que o **Mocha** formate a saída. Inclusive existem outros projetos que possuem asserções mais avançadas, como o **Should.js**.

6.2 FAZENDO REQUISIÇÕES COM O SUPERTEST

Por mais que já tenhamos melhorado nosso código de teste, ainda somos obrigados a lidar diretamente com a API de requisições do Node.js.

```
var options = {
  hostname: 'localhost',
  port:3000,
  path: '/produtos',
  headers: {
    'Accept': 'application/json',
  }
};

http.get(options, function(res) {
  assert.equal(res.statusCode,200);
  assert.equal(res.headers['content-type'],
  "application/json; charset=utf-8");

  done();
});
```

Temos que acessar diretamente as propriedades do *response* como, por exemplo, o *headers*. Mesmo na hora de enviar a requisição, temos que montar um JSON com toda nossa configuração. Pensando em facilitar este tipo de teste, foi criado uma biblioteca chamada **SuperTest**. Analise como fica o código para executarmos exatamente o mesmo teste.

```
var request = require('supertest')();

it('#listagem de produtos json', function (done) {
    request.get('http://localhost:3000/produtos')
        .set('Accept', 'application/json')
        .expect('Content-Type', /json/)
        .expect(200,done)

});
```

Carregamos o módulo do **SuperTest**, ao invés de usar diretamente a API sobre o HTTP fornecida pelo Node.js. Temos algumas vantagens nesse trecho de código.

- Não precisamos criar o JSON de configuração para fazer a requisição

- As asserções baseadas no response já estão encapsuladas em funções
- Diminuimos o número de funções anônimas necessárias.

Um detalhe bem interessante é que as funções do **SuperTest** já aceitam como argumento a função de finalização passada pelo **Mocha**! Lembre sempre de usar a função de finalização, sem ela o Mocha vai encerrar o teste antes do resultado chegar.

Deixando de subir o servidor

Para que nossos testes funcionem, até este momento, é obrigatório que o servidor sempre seja iniciado. Por mais que isso não tenha nos incomodado muito, é um ponto a mais que devemos lembrar. Como o *express* é uma biblioteca muito conhecida, o SuperTest já fornece um meio de simularmos as requisições diretamente pelo próprio *express*.

```
var express = require('../custom-express')()
var request = require('supertest')(express);

it('#listagem de produtos json', function (done) {
    request.get('/produtos')
        .set('Accept', 'application/json')
        .expect('Content-Type', /json/)
        .expect(200,done)

});
```

Simplesmente carregamos o nosso arquivo que configura o *express* e passamos objeto para o SuperTest. Ao invés de ficar passando o endereço do servidor, apenas passamos o caminho da rota já configurada. Estamos simulando a mesma requisição, só que agora não é mais necessário temos um servidor HTTP rodando.

6.3 TIPOS DE TESTES

O teste que fizemos envolveu várias camadas da nossa aplicação. O banco de dados, o *express*, validadores etc. Este tipo de teste, que realmente usa várias partes do projeto, é conhecido como **Teste de Integração**. Ele é muito usado para verificar se todas as partes do sistema estão funcionado em conjunto.

Caso tivéssemos uma lógica mais complicada, isolada em uma função, poderíamos ter realizado um teste apenas para este trecho, também conhecido como **Teste de Unidade**.

6.4 EXERCÍCIOS: ESCREVENDO TESTES PARA A APLICAÇÃO

1. Para utilizarmos o Mocha e o Supertest, precisamos instalá-los no projeto. Note que, dessa vez, o comando para salvar é um pouco diferente e ele já indica que queremos ter essas bibliotecas apenas no ambiente de desenvolvimento (`--save-dev`), mas não em produção:

```
npm install mocha --save-dev
npm install supertest --save-dev
```

2. O Mocha trabalha com o padrão de juntar todos os testes dentro da pasta `test` e, dentro dela, seguimos o mesmo padrão de pastas que encontramos na pasta `app`. *No terminal*, vá para o diretório raiz do projeto e faça:

```
mkdir -p test/routes
cd test/routes
```

3. Agora, vamos começar a escrever testes relativos à listagem de produtos. Inicialmente crie o arquivo `produtos.js` na pasta recém criada.

4. Com o arquivo criado, é necessário implementarmos o código de teste.

```
var express = require('../../custom-express')();
var request = require('supertest')(express);

describe('#ProdutosController', function() {

    it('listagem de produtos json', function (done) {
        request.get('/produtos')
            .set('Accept', 'application/json')
            .expect('Content-Type', /json/)
            .expect(200,done)

    });

    it('listagem de produtos html', function (done) {
        request.get('/produtos')
            .expect('Content-Type', /html/)
            .expect(200,done)

    });
}
```

5. Como criamos uma subpasta, rode os testes da seguinte forma:

```
node_modules/mocha/bin/mocha --recursive
```

   A opção `--recursive` faz com que o Mocha procure por testes em todos os subdiretórios.

6. Tente escrever mais um caso de teste, dessa vez para verificar se nossa lógica de cadastro está funcionando como deveria. Escreva os seguintes testes:

   - Verifique o cadastro correto de um produto e se realmente foi realizado um *redirect*
   - Caso um produto inválido seja cadastrado, verifique se o status foi retornado corretamente.

   Para descobrir como simular o envio de dados de um formulário, analise um pouco o código do próprio *supertest*. Acesse o endereço http://bit.ly/test-supertest.

6.5 DIFERENCIANDO OS AMBIENTES DE EXECUÇÃO

Até agora, para realizar nossos testes, temos usado o nosso banco de desenvolvimento. E se tivéssemos uma regra que dois produtos não poderiam ter o mesmo nome? O que ia acontecer quando fossemos executar o mesmo teste duas vezes? O nosso teste ia quebrar, já que na segunda vez já teria um produto cadastrado com o mesmo nome.

Uma boa solução para este tipo de cenário é o de limpar o banco entre cada execução de teste. Mas aí caímos em outro problema: como o banco de desenvolvimento está sendo usado, é ele que será limpo entre cada execução, removendo, por exemplo, todos os nossos dados cadastrados manualmente!

Para resolver este problema podemos usar dois bancos diferentes, um para cada ambiente. O módulo responsável por carregar nossa conexão está no arquivo *connectionFactory.js*. É justamente no momento da criação da conexão que devemos decidir para qual banco apontar.

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

function createDBConnection(){
  if (ambienteDeDev) {
    return mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: '',
      database: 'casadocodigo_nodejs'
    });
  }

  if (ambienteDeTest) {
    return mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: '',
      database: 'casadocodigo_nodejs_teste'
    });
  }
}
```

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

O único ponto que falta é descobrirmos em qual ambiente estamos. Para fazermos isso podemos passar argumentos de execução para a aplicação que vai rodar no Node.js.

```
NODE_ENV=test node_modules/mocha/bin/mocha test
```

O nome da variável pode ser qualquer um, mas no mercado se convencionou usar **NODE_ENV**. Agora podemos usar essa informação para diferenciar os ambientes.

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

function createDBConnection(){
  if (!process.env.NODE_ENV) {
    return mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: '',
      database: 'casadocodigo_nodejs'
    });
  }

  if (process.env.NODE_ENV == 'test') {
    return mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: '',
      database: 'casadocodigo_nodejs_teste'
    });
  }
}

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

Através da variável global **process**, temos acesso a um objeto com todas as informações do ambiente de execução. Por exemplo, podemos acessar as todas variáveis de ambiente do sistema operacional através da propriedade **env**. Por fim, só fazemos um teste baseado no argumento que passamos na execução e carregamos a conexão correta.

6.6 EXERCÍCIOS: CRIANDO CONEXÕES POR AMBIENTE

1. Crie um novo banco de dados chamado `casadocodigo_teste` com a mesma estrutura do banco original. Você pode seguir os mesmos passos do capítulo de acesso ao banco de dados ou simplesmente duplicar o atual fazendo os seguintes passos:

```
mysqladmin -u root create casadocodigo_teste
mysqldump -u root casadocodigo > cdc.sql
mysql -u root casadocodigo_teste < cdc.sql
rm cdc.sql
```

2. Altere o arquivo `infra/connectionFactory.js` para suportar a diferenciação de ambientes,

adicionando um simples `if` de verificação da variável de ambiente `NODE_ENV`:

```
var databaseName = "casadocodigo";
if (process.env.NODE_ENV == 'test') {
    databaseName = 'casadocodigo_teste';
}
```

6.7 PARA SABER MAIS: LIMPANDO O BANCO ENTRE OS TESTES

Na seção anterior comentamos que seria interessante ter sempre o banco limpo entre nossos testes. O objetivo é que os dados manipulados por um teste não influencie no resultado do outro. Basicamente o que precisamos fazer é executar as queries de *delete* antes de cada teste nosso. Se tivermos uma função `limpaTabelas` queremos rodá-la antes de fazer os requests dos testes.

Considerando a forma de lidar com callbacks que você já está acostumado, à essa altura, teremos algo parecido com o que segue:

```
it('#listagem de produtos json', function (done) {
    limpaTabelas(function(){
        request.get('/produtos')
                .set('Accept', 'application/json')
                .expect('Content-Type', /json/)
                .expect(200,done);
    });
});

it('#cadastro de um novo produto com tudo preenchido', function (done) {
    limpaTabelas(function(){
        request.post('/produtos')
                .send({titulo:"novo livro",
                        preco:20.50,
                        descricao:"livro de teste"})
                .expect(302)
                .end(function(err,response){
                    done();
                });
    });

});
```

Simplesmente criamos uma função para limpar as tabelas do banco de dados e, como tudo é executado de maneira assíncrona, somos obrigados a passar uma função de *callback* que deve ser executada após a query ser realizada. O código é até simples, o problema é que o tempo inteiro vamos ter que ficar chamando essa função, sem contar a complexidade adicionada pelo *callback* extra. Para facilitar o nosso trabalho, o **Mocha** permite a criação de uma função, com um nome específico, que é chamada antes de cada teste.

```
describe('#ProdutosController', function() {

    var limpaTabelas = function(done) {
        var conn = express.infra.connectionFactory();
        conn.query("delete from livros",function(ex,result){
            if(!ex) {
                done();
```

```
            }
        });
    }

    beforeEach(function(done) {
        limpaTabelas(done);
    });

    ...
});
```

Além de fazer alguma coisa antes de cada teste, podemos executar algum código depois de cada função, ou até mesmo antes de todas. Veja as possibilidades.

- before ; Permite que um código rode antes de todos os testes.
- after ; Permite que um código rode depois de todos os testes.
- afterEach ; Permite que um código rode depois de cada teste.

E se fossem muitas tabelas?

Caso a aplicação utilize muitas tabelas o nosso código de limpar o banco vai começar a ficar um tanto complexo. Seríamos obrigados a deletar os dados de todas elas e o nosso arquivo de testes começaria a ficar muito mais complicado do que deveria.

Pensando em simplificar isso, foi criada a biblioteca *node-database-cleaner*. A ideia é que ela limpe todas as tabelas associadas a nossa conexão com o banco de dados.

```
var express = require('../../custom-express')()
var request = require('supertest')(express);
var DatabaseCleaner = require('database-cleaner');

describe('#ProdutosController', function() {

    beforeEach(function(done) {
        var databaseCleaner = new DatabaseCleaner('mysql');
        databaseCleaner.clean(express.infra.connectionFactory(), done);
    });

    after(function(done) {
        var databaseCleaner = new DatabaseCleaner('mysql');
        databaseCleaner.clean(express.infra.connectionFactory(), done);
    });

...
```

O objeto do tipo `DatabaseCleaner` é o responsável por todo trabalho. O método `clean` recebe o objeto que representa a nossa conexão com o banco e, como não poderia faltar, um *callback* que deve ser chamado quando a tarefa acabar. A *lib* ainda suporta integração com outros tipos de bancos, relacionais ou não.

- PostgreSQL;
- MongoDB

6.8 EXERCÍCIOS OPCIONAIS

1. Instale o *node-database-cleaner* através do comando `sudo npm install database-cleaner --save-dev` .
2. Altere o código do nosso teste para usar o objeto `DatabaseCleaner` , em vez de controlar os dados da tabela de teste na mão.

CAPÍTULO 7

ENVIANDO E RECEBENDO INFORMAÇÕES VIA WEBSOCKET

Uma funcionalidade desejada pela loja é a de poder avisar os clientes sobre promoções relâmpago. A ideia é que o administrador possa decidir que algum livro fique em promoção em determinado instante e que todos os usuários que estejam em alguma página, naquele momento, sejam notificados.

7.1 CONSTRUINDO A HOME DA CASA DO CÓDIGO

Para que possamos notificar os usuários, primeiro é necessário que tenhamos pelo menos a *home* da Casa do Código. Na verdade é um tipo de código que já trabalhamos bastante durante o treinamento, vamos precisar de uma página e também mapear uma rota que leve o usuário para essa página. Algo como o que segue:

```
module.exports = function(app) {
    app.get("/",function(req, res) {
        var connection = app.infra.connectionFactory();
        var produtos = new app.infra.ProdutoDao(connection);

        produtos.lista(function(error,results,fields){
            res.render('home/index',{livros:results});
        });
        connection.end();

    });
}
```


7.2 EXERCÍCIOS: CONSTRUINDO A HOME DA CASA DO CÓDIGO

1. Primeiro precisamos de todos os arquivos de estilo relacionados a home do site, além do conteúdo base da página em si. Siga os seguintes passos:

   - no seu Desktop, clique na pasta que leva para os cursos da **Caelum**;
   - entra na pasta `JS-35/casadocodigo`
   - copie a pasta `site` para a pasta `public/css` da sua aplicação;
   - copie o arquivo `index.ejs` para a pasta `views/home` ;

2. No arquivo `views/home/index.ejs` está marcado os pontos onde devemos executar o código dinâmico. Os trechos devem ficar iguais aos que seguem:

```
<%for(i=0; i<3; i++) {%>
<li class="col-left">
    <a href="linkDetalhe" class="block clearfix">
        <img width="170" height="240" src="urlImagem"
        alt="<%=livros[i].titulo%>"
        title="<%=livros[i].titulo%>"/>

        <h2 class="product-title"><%=livros[i].titulo%></h2>
        <small class="buy-button">Lan&#231;amento!</small>
    </a>
</li>
<%}%>

<%for(i=0; i<livros.length; i++) {%>
    <li>
        <a href="linkDetalhe" class="block clearfix">
            <h2 class="product-title"><%=livros[i].titulo%></h2>
            <img width="143"
                 height="202"
                 src="linkImagemCapa"
                 alt="<%=livros[i].titulo%>"
                 title="<%=livros[i].titulo%>"/>

            <small class="buy-button">Compre</small>
        </a>
    </li>
<%}%>
```

3. Agora crie o novo arquivo para mapear as rotas referentes ao site. Ele deve ficar em `routes/site.js` . Dentro dele, vamos mapear a rota referente ao root da nossa aplicação.

```
module.exports = function(app) {
    app.get("/",function(req, res) {
        var connection = app.infra.connectionFactory();
        var produtos = new app.infra.ProdutoDao(connection);

        produtos.lista(function(error,results,fields){
            res.render('home/index',{livros:results});
        });
        connection.end();

    });
}
```

4. Suba o servidor e acesse o endereço http://localhost:3000.

7.3 COMO NOTIFICAR OS USUÁRIOS?

Agora que temos a home da nossa aplicação, a pergunta que fica na nossa cabeça é: como vamos enviar alguma coisa para página que está sendo acessada pelo usuário, neste instante. Uma das soluções mais antigas é implementar um JavaScript que fica, de tempos em tempos, consultando uma URL do servidor para saber se alguma nova aconteceu.

```
//exemplo usando um pouco de jquery
setInterval(function(){
    $.ajax({
        url: "/novas/promocoes",
        success: function(data){
            //Atualiza o valor de cada ação na tabela
    }, dataType: "json"});
}, 30000);
```

O grande problema dessa abordagem é que o cliente fica o tempo inteiro disparando requisição contra o servidor, mas na maioria das vezes não tem nada de novo acontecendo! Basicamente estamos metralhando o servidor com várias requisições inúteis.

7.4 API DE WEBSOCKETS E O NAVEGADOR

Na verdade, o que precisamos é que quando alguma coisa aconteça no servidor, ele notifique o cliente com a novidade. Para contornar o problema de ficar fazendo requisição o tempo inteiro, ainda antigamente, os servidores começaram a suportar que certas conexões abertas pelo cliente não fossem fechadas, técnica conhecida como Comet. Essa foi uma técnica criada justamente para os servidores começarem a suportar o chamado **Long Polling** e, dessa forma, que o servidor pudesse enviar informações para o cliente, sem a necessidade de uma nova requisição.

Pois bem, na última versão do HTML5 esse tipo de técnica virou uma especificação chamada de WebSocket. Só que eles foram além de só especificar o que já tinha pronto e adicionaram novos detalhes.

- Além do servidor poder notificar o cliente, o cliente também pode enviar informações para o servidor.
- A comunicação é feita baseada em um novo protocolo, específico para o WebSocket.
- API padrão para ser usada dentro do navegador, ao invés de ficar simulando requisição AJAX.

Usando a API de WebSockets no navegador

Vamos começar a implementar nossa nova funcionalidade. A primeira coisa que precisamos é abrir o WebSocket na página, para podermos mandar e receber informações pelo canal.

```
var ws = new WebSocket("ws://localhost:3000");
```

Criamos um novo objeto do tipo **WebSocket** passando justamente o endereço que que vamos manter a conexão aberta com o cliente. Agora que a conexão está aberta, precisamos receber as novas

mensagens, que serão enviadas a partir do servidor.

```
ws.onmessage = function(message) {
    if(message.type == 'novaPromocao'){
        document.location.href= "/produtos/"+data.livro.id+"
            ?promocao="+data.mensagem;
    }
}
```

A propriedade *onMessage* aceita um função, que vai ser invocada sempre que o servidor enviar uma nova mensagem para o cliente. Perceba que a nossa função simplesmente leva o usuário para a tela de detalhe do livro com o *id* igual ao que foi recebido como argumento. A API de WebSockets é muito direta, a abstração criada pela especificação realmente deixou tudo muito simples. Um outro detalhe importante é a verficação do tipo da mensagem, já que podemos ter vários tipos de mensagens sendo enviadas para os mais variados cenários.

Também é interessante notar o console do navegador. Quando acessamos uma página que abre um WebSocket, a requisição fica aberta com o status 101, que representa a conexão em andamento.

Caso você use o serviço do WhatsApp WEB, também abra o console da página e perceba que ele usa um WebSocket para ficar enviando e recebendo novas mensagens.

7.5 WEBSOCKETS COM SOCKET.IO

Agora que nosso cliente já está pronto, é necessário que seja implementada a parte do sevidor. O Node.js não tem nada pronto para lidarmos com requisições baseadas em WebSockets e, para não termos que lidar com uma API de muito baixo nível, vamos usar um biblioteca bem consolidada no mercado, o Socket.io.

Como já é de praxe, precisamos instalar o módulo no nosso projeto.

```
npm install socket.io --save
```

Agora é necessário termos uma tela que deve ser acessada pelo administrador da loja, de modo que ele consiga colocar um livro em promoção. Vamos criar o arquivo em *views/promocoes/form.ejs*

```
<html>
    <body>
        <form action="/promocoes" method="post">
            <div>
                <input type="text" name="mensagem"/>
            </div>
            <div>
                <select name="livro[id]">
                    <% for(var i=0; i<lista.length; i++) {%>
                        <option value="<%=lista[i].id%>">
                            <%=lista[i].titulo%></option>
                    <% } %>
                </select>
            </div>
            <input type="submit" value="Promoção relâmpago"/>

        </form>

    </body>
</html>
```

Para acessar essa tela, vamos usar o endereço **/promocoes/nova**. Como já fizemos antes, vamos precisar registrar uma rota e associar uma função para tratar ela. Vamos focar na função, que é de onde teremos de notificar os clientes sobre a nova promoção relâmpago.

```
    app.post("/promocoes", function(req,res) {
        var promocao = req.body;

        //precisamos disparar a notificacao

        res.redirect("/promocoes/form");
    });
```

Perceba que recebemos as informações referente ao produto em questão, só precisamos enviar a mensagem para o navegador dos clientes.

```
    controller.salva = function(req,res) {
        var promocao = req.body;

        socketIo.emit("novaPromocao",promocao);

        res.redirect("/promocoes/form");
    };
```

A função *emit* recebe como primeiro argumento o tipo de evento que queremos enviar para os clientes. Além disso passamos um JSON com o conteúdo da mensagem em si. Um último ponto que ainda ficou sem explicação é: como o objeto do Socket.io ficou disponível para ser usado pelas funções do nosso *controller*? Inicialmente precisamos criar uma nova instância do Socket.io e, além disso, associar ele a uma instância de Server, uma classe da API do Node.js.

```
    var http = require('http').Server(app);
    var io = require('socket.io')(http);
```

E aqui temos um problema. A instância do Server precisa ser a mesma que usamos para colocar o servidor no ar, só que esse código está no arquivo *app.js*.

```
var app = require('./custom-express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

var server = http.listen(3000, function () {

    var host = server.address().address;
    var port = server.address().port;

    console.log('Example app listening at http://%s:%s', host, port);

});
```

Agora precisamos encontrar uma maneira de pegar a instância do Socket.io e deixar disponível dentro dos controllers. Para a nossa sorte, o objeto que guarda a referência para o *express*, possui o método *set*, que inclusive já usamos. Podemos usá-lo para associar um objeto a uma chave.

```
var app = require('./custom-express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.set('io',io);

var server = http.listen(3000, function () {

    var host = server.address().address;
    var port = server.address().port;

    console.log('Example app listening at http://%s:%s', host, port);
});
```

Dessa forma, quando um *controller* precisar lidar com um WebSocket, será necessário apenas recuperar o objeto que está no *express*.

```
controller.salva = function(req,res) {
    var promocao = req.body;
    app.get('io').emit("novaPromocao",promocao);
    res.redirect("/promocoes/form");
};
```

7.6 WEBSOCKETS NO CLIENTE

Para conectar via WebSocket com o nosso servidor, utilizamos diretamente a API de WebSockets disponível nos navegadores. Por mais que isso seja possível, conseguimos simplificar bastante o código utilizamos o Socket.io no lado do cliente.

```
<script src="/socket.io/socket.io.js"></script>
<script>
    var socket = io();
    socket.on('novaPromocao', function (data) {
           document.location.href=
               "/produtos/"+data.livro.id+"?promocao="+data.mensagem;
    });
```

```
</script>
```

Aqui precisamos de um pouco de tempo para analisar o código. Importamos o script de uma URL que não configuramos em nenhum lugar. Não colocamos nenhum arquivo na pasta *public* e nem configuramos uma rota para um controller. O Socket.io no lado do servidor se integra com o Node.js e responde para a rota */socket.io/socket.io.js*. Além disso o Socket.io além de suportar os WebSockets, também suporta as outras maneiras de receber dados do servidor, por exemplo, *pooling*. A ideia é abstrair a forma e usar a que estiver disponível no navegador.

Uma última facilidade é a função *on*. Ela já recebe o tipo do evento que você quer escutar, tirando a necessidade de ficar fazendo *ifs* no lado do cliente.

7.7 EXERCÍCIOS: NOTIFICANDO OS CLIENTES SOBRE PROMOÇÕES

1. Instale o Socket.io no seu projeto.

```
npm install socket.io --save
```

2. Precisamos fazer a tela de cadastro de promoções. Primeiramente vamos criar as funções que tratam as rotas. Crie o arquivo `routes/promocoes.js` e dentro dele adicione o seguinte código.

```
module.exports = function(app) {
    app.get("/promocoes/form", function(req,res) {
        var connection = app.infra.connectionFactory();
        var produtoDao = new app.infra.ProdutoDao(connection);

        produtoDao.lista(function(error,results){
            res.render('promocoes/form',{lista:results});
        });

    });

    app.post("/promocoes", function(req,res) {
        var promocao = req.body;

        //vamos disparar a notificação
        res.redirect("/promocoes/form");
    });
}
```

3. Agora precisamos criar a página de cadastro de novas promoções. Crie o arquivo `views/promocoes/form.ejs` e dentro dele adicione o seguinte código.

```
<html>
    <body>
        <form action="/promocoes" method="post">
            <div>
                <input type="text" name="mensagem"/>
            </div>
            <div>
                <select name="livro[id]">
                    <% for(var i=0; i<lista.length; i++) {%>
                        <option value="<%=lista[i].id%>">
                            <%=lista[i].titulo%>
```

```
                        </option>
                    <% } %>
                </select>
            </div>
            <input type="submit" value="Promoção relâmpago"/>

        </form>

    </body>
</html>
```

4. Agora que cadastramos novas promoções, vamos disparar o evento. Vamos alterar o código da função que recebe a nova promoção.

```
app.post("/promocoes", function(req,res) {
    var promocao = req.body;

    app.get('io').emit("novaPromocao",promocao);
    res.redirect("/promocoes/form");
});
```

5. É necessário que o objeto do Socket.io fique disponível para que possa ser acessado de dentro da rota. Altere o arquivo `server.js` para que ele fique parecido com o seguinte.

```
var app = require('./custom-express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.set('io',io);

var server = http.listen(3000, function () {

    var host = server.address().address;
    var port = server.address().port;

    console.log('Servidor executando em http://%s:%s',
        host, port);

});
```

6. Por fim, precisamos adicionar o código Javascript do lado cliente, na página da home do site. Coloque ele no fim da página, logo antes do fechamento da tag `</body>`.

```
<script src="/socket.io/socket.io.js"></script>
<script>
    var socket = io();
    socket.on('novaPromocao', function (data) {
        console.log("Nova promoção");
        console.log(data);
    });
</script>
```

CAPÍTULO 8

MIDDLEWARES NO EXPRESS

No arquivo *custom-express.js* usamos mais de uma vez o método `use` presente no módulo do express. Ele foi utilizado para incorporarmos algumas funcionalidades dentro da nossa aplicação, por exemplo:

- Parser do conteúdo do formulário;
- Validação dos dados da requisição;
- Liberação de acesso a conteúdo estático da aplicação;

Basicamente esse método permite que passemos uma função que deve ser executada antes e(ou) depois do request chegar efetivamente a alguma rota da nossa aplicação.

ENTENDENDO OS MIDDLEWARES

Vamos dar uma olhada no que é retornado quando invocamos a função `expressValidator()` e passamos ela para o método `use` .

```
return function(req, res, next) {
  var locations = ['body', 'params', 'query'];

  req._validationErrors = [];
  req.validationErrors = function(mapped) {
    if (mapped && req._validationErrors.length > 0) {
      var errors = {};
      req._validationErrors.forEach(function(err) {
        errors[err.param] = err;
      });

      return errors;
    }

    return req._validationErrors.length > 0 ? req._validationErrors : false;
  };

  ....

  req.filter = req.sanitize;
  req.assert = req.check;
  req.validate = req.check;

  next();
};
```

Esse código revela para a gente muita coisa. Perceba que ele retorna uma função que recebe os seguintes argumentos: request, response e um **next**. Já vamos ver a utilidade do **next** em alguns minutos.

Os dois primeiros já estamos mais que acostumados, já que são os mesmos que recebemos quando tratamos as requisições associadas a nossa rota. Um outro detalhe interessante é que a função retornada, quando invocada, altera a estrutura do objeto que representa o request, adicionando novos comportamentos no mesmo.

```
req.validationErrors = function(mapped) {
    ...
    var errors = {};
    req._validationErrors.forEach(function(err) {
      errors[err.param] = err;
    });
}
```

A função `validationErrors` foi a que nós invocamos de dentro da nossa rota para saber se os dados tinham sido validados corretamente. Por sinal uma parte da implementação até mostra como ele guarda os erros no JSON que passamos para nossa view. Quando passamos essa função para o método `use`, estamos solicitando ao express que execute a mesma durante o fluxo de tratamento de uma requisição. O momento que ela vai ser executada depende de onde você invocar. Perceba que invocamos ela depois da configuração do **body-parser** e antes da configuração das nossas rotas.

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

require('./routes/produtos')(app);
```

A ideia é que os valores já cheguem nela parseados. Só que precisamos invocar antes da nossa rota, já que na nossa função de tratamento de request, já fazemos uso dos métodos que foram adicionados no request. Esse tipo de função é chamada de **middleware**, pois o objetivo dela é que possamos interceptar o request e adicionar verificações e comportamentos sobre ele. Muitos dos plugins que usamos junto com o express fazem uso dos middlewares!.

Já que vários middlewares podem ser executadas durante uma requisição, também recebemos como argumento uma função responsável por chamar o próximo da fila. É justamente para isso que serve o parâmetro *next*. No fim da execução da sua função, se for do seu interesse, você pode pedir para o express continuar o fluxo.

```
return function(req, res, next) {
    ...

    //aqui estamos pedindo para continuar
  next();
};
```

É muito importante que você invoque a função que representa o próximo passo, caso contrário o processamento vai para nesse middleware. Caso você esteja desenvolvendo algum mecanismo de proteção contra acesso não autorizado, aí pode fazer sentido não continuar o fluxo.

CRIANDO NOSSO PRÓPRIO MIDDLEWARE

Nesse momento, quando um cliente solicita um recurso que não foi encontrado pela nossa aplicação, é retornado para ele uma página feia com uma mensagem gerada pelo próprio express.

```
Cannot GET /enderecoInexistente
```

Para resolvermos isso, podemos adicionar um middleware responsável por direcionar o usuário para uma página mais amigável.

```
app.use(function(req, res, next){
    res.status(404).render("erros/404");
});
```

Aqui entra a questão da ordem dos middlewares. Caso a gente adicione a função antes do carregamento das nossas rotas, para toda requisição será retornado o status

1. Para tudo continuar funcionando corretamente, precisamos adicionar a função após o carregamento das nossas rotas.

```
require('./routes/produtos')(app);

app.use(function(req, res, next){
    res.status(404).render("erros/404");
});
```

8.1 MIDDLWARE PARA ERROS DA APLICAÇÃO

Nenhuma aplicação está livre de erros inesperados. Para esses casos, o mais comum é retornar o status 500 para o cliente em questão. O express possui um handler especial para esses casos.

```
app.use(function(error,req, res, next){
    res.status(500).render("erros/500");
});
```

Perceba que recebemos como primeiro argumento o erro em questão, justamente para o express saber que esse middleware tem que ser chamado apenas em casos de exceptions lançadas pela aplicação. Lembre que, mais uma vez, ele precisa ser adicionada após a configuração das nossas rotas, já que se ele for adicionado antes a função vai não vai ser invocada porque nenhum erro terá acontecido.

8.2 EXERCÍCIOS: CRIANDO OS MIDDLEWARES PARA TRATAMENTO DE ERROS.

1. Crie a página de erro para lidar com os recursos não encontrados. Cria ela dentro de *views/erros*. Além disso adicione um middleware para lidar com tal problema.

2. Crie a página de erro para lidar com os erros inesperados. Cria ela dentro de *views/erros*. Além disso adicione um middleware para lidar com tal problema.

3. Desafio: Nossos middlewares de tratamentos de erro só respondem páginas. O que funciona bem para aplicações que só acessadas por navegadores. Como esse não é nosso caso, faça com que os middlewares suportem também requisições que querem JSON como resposta.

Capítulo 9

DEPLOY DA APLICAÇÃO

Já implementamos tudo que era necessário para termos a versão inicial da nossa aplicação. Só que até este momento estamos executando tudo em nossa própria máquina. O objetivo agora é que a gente pegue a nossa aplicação e instale em um servidor que pode ser acessado por todos os nossos clientes, o famoso processo de deploy.

ONDE REALIZAR O DEPLOY?

Como foi visto durante todo treinamento, a infraestrutura básica para executar uma aplicação em cima do Node.js não é muito complexa. Precisamos ter o runtime dele instalado e pronto. Por exemplo, caso sua empresa possua uma máquina específica para o deploy, basta que a aplicação seja copiada para lá e, do terminal, você execute o comando `node app.js`.

É importante sempre ter isso em mente, rodar a aplicação no ambiente de produção não difere muito de rodar ela em desenvolvimento. O que geralmente muda são as configurações da máquina, dados de acesso aos serviços e, possivelmente, você não vai querer que todas as dependências sejam instaladas. Por exemplo, por que nós vamos instalar o **mocha** no nosso ambiente de produção? Para resolver isso, basta que ao invés rodar `npm install`, a gente execute o mesmo comando passando o argumento `--production`.

Uma outra opção de deploy, que pelo menos inicialmente deixa tudo muito mais fácil, é utilizar as plataformas prontas que automatizam o process de deploy e host, os já conhecidos clouds.

HEROKU

Uma das opções de cloud para deploy de aplicações escritas em JavaScript que rodam sobre o Node.js é o Heroku. Para conseguirmos realizar o deploy no ambiente deles precisamos apenas seguir alguns passos.

9.1 CRIAÇÃO DE UM USUÁRIO E INSTALAÇÃO DO UTILITÁRIO

A primeira tarefa é criar um usuário na plataforma. O processo é simples, sendo todo feito através do site deles, o heroku.com. Com o usuário criado, o próximo passo é instalar um programa de linha

comando oferecido pelo heroku que automatiza todo o processo de criação de novas apps, instalações de infraestrutura como um banco de dados e muito mais.

O utilitário, também chamado de Heroku Toolbelt, pode ser baixado no endereço https://toolbelt.heroku.com/. Existem versões para várias plataformas, escolha a que se adequar ao seu ambiente de desenvolvimento.

9.2 CRIAÇÃO DA INFRAESTRUTURA PARA APLICAÇÃO

Com o toolbelt instalado, precisamos criar uma aplicação que vamos usar justamente para enviar o nosso código. Nesse exato momento, no terminal do seu computador, você já deve poder executar um programa de linha comando chamado **heroku**. Por exemplo, para nos logarmos e podermos realizar várias operações, podemos executar o seguinte comando:

```
heroku login
```

Após ter realizado o login, conseguimos executar vários outros comandos. Um primeiro e necessário é justamente o comando que vai criar a aplicação para a gente.

```
heroku apps:create casadocodigo
```

Também é necessário que tenhamos o banco de dados instalado, e no nosso caso precisamos do MySQL. Aqui começar a brilhar ainda mais o Heroku, conseguimos adicionar um novo banco de dados na nossa aplicação através do próprio toolbelt.

```
heroku addons:create cleardb:ignite
```

O ClearDB é o serviço usado pelo heroku para criação de bancos de dados MySQL. Esse serviço é um cloud específico para banco de dados que, para a nossa sorte, o Heroku já tem integração total. Um outro detalhe interessante foi o tipo de plano de banco de dados que escolhemos, no caso o **ignite**. Esse é o plano mais básico oferecido pelo Heroku, onde temos apenas algumas megas de espaço para salvar dados, se for necessário mais que isso, já é necessário pagar.

Nesse momento já temos quase tudo que é necessário para rodarmos nossa aplicação, inclusive até já podemos saber qual é o endereço do nosso banco de dados. Só precisamos executar o comando `heroku config`. Abaixo segue um exemplo de saída:

```
CLEARDB_DATABASE_URL:    mysql://bae2fca1ed0ced:c41afecd@us-cdbr-iron-east-03.
                            cleardb.net/heroku_e4cbb8fc2c1fee8?reconnect=true
HEROKU_APP_ID:           17df0971-16fb-4e53-8029-97c12c27f5de
HEROKU_APP_NAME:         casadocodigo-nodejs
```

Essas são todas variáveis de ambiente disponíveis na sua máquina criada lá no ambiente do Heroku. A primeira delas é justamente o endereço de acesso para sua instância do banco de dados.

9.3 CRIANDO AS TABELAS NO BANCO DE DADOS REMOTO

Ainda vamos enviar nosso código para o heroku, mas uma pergunta que já pode ser feita é: em quais tabelas vamos gravar nossos dados? Perceba que criamos apenas o banco de dados, mas ainda não criamos nenhuma tabela no ambiente remoto. Para fazer isso, precisamos acessar o MySQL criado pelo Heroku e rodar o nosso script de criação da tabela.

```
mysql -u bae2fca1ed0ced -h us-cdbr-iron-east-03.cleardb.net -p
```

Executamos o comando `mysql` só que apontamos para o endereço remoto e, além disso também informamos o usuário remoto que foi criado. Todas essas informações estão contidas no endereço de acesso que está guardado na variável de ambiente.

```
mysql://bae2fca1ed0ced:c41afecd@us-cdbr-iron-east-03.cleardb.net/
    heroku_e4cbb8fc2c1fee8?reconnect=true
```

O trecho acima é só um exemplo, mas já serve de base. O login e a senha estão separados pelo caracter **:**, `bae2fca1ed0ced:c41afecd`. Depois vem justamente o endereço do servidor do banco de dados e, por último, o nome do banco que foi criado para a gente. Depois de acessar o servidor, é necessário que informemos qual database precisamos utilizar.

```
use nomeDoBancoCriadoParaVocê
```

Dentro do banco de dados correto, só precisamos rodar o script de criação da tabela.

```
CREATE TABLE livros (
  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  titulo varchar(255) DEFAULT NULL,
  descricao text,
  preco decimal(10,2) DEFAULT NULL);
```

Pronto! Agora já temos toda infraestrutura preparada para que possamos deployar nossa aplicação no Heroku.

PREPARANDO A APLICAÇÃO

Com toda a infraestrutura pronta, chegou a hora de analisarmos a nossa aplicação para sabermos se

ela já pode ser deployada.

9.4 CONFIGURAÇÃO DE ACESSO AO BANCO DE DADOS

Nesse exato momento, o nosso arquivo **connectionFactory.js** contém o seguinte código.

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

function createDBConnection(){
    if (!process.env.NODE_ENV) {
        return mysql.createConnection({
            host: 'localhost',
            user: 'root',
            password: '',
            database: 'casadocodigo_nodejs'
        });
    }

    if (process.env.NODE_ENV == 'test') {
        return mysql.createConnection({
            host: 'localhost',
            user: 'root',
            password: '',
            database: 'casadocodigo_nodejs_teste'
        });
    }
}

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

Ele sempre tenta conectar no banco de desenvolvimento caso a variável de ambiente `NODE_ENV` não esteja definida e, caso ela esteja, é verificado se o ambiente é o de testes. Só que quando rodamos nossa aplicação no Heroku, essa mesma variável de ambiente está definida para **production**! E agora, o que precisamos fazer? Como você deve ter imaginado, é necessário adicionar mais uma condição. Por sinal, definir a variável `NODE_ENV` com **production** é o padrão de mercado, para indicar que estamos em ambiente de produção. O código alterado deve ficar da seguinte forma:

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

function createDBConnection(){
    if (!process.env.NODE_ENV || process.env.node === 'dev') {
        return mysql.createConnection({
            host: 'localhost',
            user: 'root',
            password: '',
            database: 'casadocodigo_nodejs'
        });
    }

    if (process.env.NODE_ENV == 'test') {
        return mysql.createConnection({
            host: 'localhost',
            user: 'root',
            password: '',
            database: 'casadocodigo_nodejs_teste'
```

```
        });
    }

    if (process.env.NODE_ENV == 'production') {
        //o que vamos passar de configuração de conexão?
        return mysql.createConnection({
            host:...,
            user:...,
            password:...,
            database:...
        });
    }
}

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

Agora que já verificamos o ambiente, a outra pergunta que fica é: e quais informações de acesso serão passadas, para que possamos criar a conexão? Como já vimos, todas elas estão contidas na variável de ambiente `NODE_ENV` , então basta que façamos uso da mesma.

```
...
if (process.env.NODE_ENV == 'production') {
    var url = process.env.CLEARDB_DATABASE_URL;
    var grupos = url.match(/mysql:\/\/(.*):(.*)@(.*)\/(.*)\?/);
    return mysql.createConnection({
        host:grupos[3],
        user:grupos[1],
        password:grupos[2],
        database:grupos[4]
    });
}
```

Apenas usamos uma expressão regular para quebrar os grupos de informações necessárias para a gente. Cada informação está numa posição específica do array. Dessa forma, não precisamos deixar exposto coisas como a senha de acesso ou o login do usuário do banco de dados. Sem contar que esse código serve para qualquer aplicação sua que se conecte a algum banco de dados configurado através do heroku.

9.5 SCRIPT DE EXECUÇÃO DA APLICAÇÃO

Para rodarmos a nossaa aplicação temos executado o comando `node app` . Quando enviarmos essa mesma aplicação para o heroku, como ele vai saber que é para executar este exato comando? Para a nossa sorte, já existe uma padronização para esta situação.

Quando rodamos o comando `npm init` , lá no inicio do projeto, foi criado um arquivo chamado **package.json**. Inclusive já olhamos um pouco para ele, pois todas as dependências que instalamos estão listadas por lá.

```
{
  "name": "casadocodigo",
  "version": "1.0.0",
  "description": "app de controle de cadastro de produtos",
  "main": "app.js",
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "1.13.1",
    "ejs": "2.3.1",
    "express": "4.12.4",
    "express-validator": "2.12.1",
    "mocha": "2.2.5",
    "mysql": "2.7.0",
    "socket.io": "1.3.6"
  },
  "devDependencies": {
    "should": "^7.0.2",
    "supertest": "^1.0.1",
    "database-cleaner": "0.10.0",
    "db-migrate": "0.9.16"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }

}
```

Perceba que o json usado para configurar tem uma chave cujo nome é **scripts**. Uma das formas do Heroku executar a nossa aplicação, é executando o comando `npm start` . Nesse momento, se fizermos este teste dentro da pasta da nossa aplicação, temos uma saída parecida com a que segue:

```
Darwin 14.4.0
npm ERR! argv "/Users/alberto/.nvm/versions/node/v4.1.2/bin/node"
    "/Users/alberto/.nvm/versions/node/v4.1.2/bin/npm" "start"
npm ERR! node v4.1.2
npm ERR! npm  v2.14.4

npm ERR! missing script: start
npm ERR!
npm ERR! If you need help, you may report this error at:
npm ERR!     <https://github.com/npm/npm/issues>

npm ERR! Please include the following file with any support request:
```

O npm está reclamando que não encontrou nenhum **script** de start para a nossa aplicação. Para resolvermos isso, basta que adicionemos uma nova chave no package.json informando justamente esse o que deve ser rodado para esse comando.

```
{
  ...
  "dependencies": {
      ...
  },
  "devDependencies": {
      ...
  },
  "scripts": {
    "start": "node app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  }

}
```

Agora, caso você rode o mesmo comando, será executado o que você especificou no arquivo. Como já dissemos, o Heroku vai usar a mesma abordagem. Um outro fator importante é que a versão do Node.js que usamos localmente, seja também usada no ambiente de produção. Para garantir isso, podemos adicionar mais uma chave no *package.json* e informar a versão do Node necessária.

```
{
  ...
  "dependencies": {
      ...
  },
  "devDependencies": {
      ...
  },
  "engines": {
    "node": "4.1.1"
  },
  "scripts": {
    "start": "node app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  }
```

```
}
```

Por sinal, você deve prestar atenção nas versões de tudo que usamos durante a construção do nosso projeto. Durante o curso a gente nunca especificou a versão de nenhuma lib que foi utilizada, o que implica que usamos sempre a última versão disponível. Pensando logicamente, em geral a última versão disponível deveria ser sempre a utilizada, já que é a que possui o maior número de correções e tudo mais. Entretanto, o seu projeto pode depender de versões específicas de libs, então fique atento a tudo isso.

9.6 CONTROLANDO A VERSÃO COM O GIT

Agora que já alteramos tudo que era necessário também na aplicação, falta apenas o último passo, que é habilitarmos o git no nosso projeto. Todo o processo de deploy do Heroku é baseado no sistema de controle de versão git. A primeira tarefa é justamente habilitar o controle dentro do projeto. Dentro da pasta execute o seguinte comando:

```
git init
```

A partir deste momento o git pode controlar todas as pastas e arquivos que tem dentro do seu projeto, só tem um problema, a pasta *node_modules*. Ela contém todas as libs que foram usadas durante a construção da aplicação, só que precisamos lembrar que o Node.js é instalado em função do sistema operacional que cada um roda. Caso pessoas do mesmo time possuam máquinas diferentes, a instalação do Node vai mudar e , consequentemente, algumas libs podem parar de funcionar. Para garantir que você não vai sofrer com isso, é uma boa prática ignorar essa pasta do controle do git. Para isso, basta que a gente crie um arquivo chamado **.gitignore** e adicione tudo que precisa ser ignorado lá dentro.

```
node_modules
```

Agora precisamos comitar tudo que fizemos, para que o git possa guardar o histórico do nosso trabalho.

```
git add .
git commit -m "commit de deploy"
```

Com tudo controlado, precisamos enviar o nosso código para o repositório git que foi criado para a gente, lá no ambiente do heroku. Quando o comando `heroku apps:create` foi executado, o utilitário do heroku já adicionou esse repositório remoto para a gente. Você pode dar uma olhada executando o seguinte comando:

```
git remote
```

Agora basta enviar o código com `git push heroku master` . Quando você enviar o código para o heroku, vai aparecer no seu console tudo que está feito remotamente em função do seu *push*. Preste bastante atenção, porque seu primeiro deploy está sendo realizado! No fim, vai aparecer o endereço de acesso da sua aplicação, copie ele no navegador e verifique se tudo está funcionando. Lembre de primeiro cadastrar alguns produtos, já que nossa página inicial assume no mínimo três produtos criados.

9.7 EXERCÍCIOS: REALIZANDO O DEPLOY

1. Baseado no texto do capítulo, tente realizar o processo de deploy no heroku.