Construindo APIs testáveis com

Waldemar Neto

Construindo APIs testáveis com Node.js

Waldemar Neto

Esse livro está à venda em http://leanpub.com/construindo-apis-testaveis-com-nodejs

Essa versão foi publicada em 2017-05-15

Tweet Sobre Esse Livro!

Por favor ajude Waldemar Neto a divulgar esse livro no Twitter!

O tweet sugerido para esse livro é:

Livro: Construindo APIs testáveis com Node.js

A hashtag sugerida para esse livro é #nodejs.

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

https://twitter.com/search?q=#nodejs

Conteúdo

Prefácio

em breve.

Introdução

O *javascript* é a uma das linguagens atuais mais populares entre os desenvolvedores (segundo o *Stack Overflow Survey* de 2017)[1]. Sorte de quem pode trabalhar com *javascript* ou está entrando em um projeto onde terá a oportunidade de aprender essa linguagem. O *javascript* é dono de uma fatia ainda única no mercado, pois é uma linguagem criada para *browsers*, para atualizar conteúdo dinamicamente, para ser não bloqueante, permitindo assim que ações sejam realizadas enquanto outras ainda estão sendo processadas. O contexto dos *browsers* contribuiu para que o *javascript* evoluísse de uma forma diferente das demais linguagens, focando em performance e em possibilitar a criação de interfaces com uma melhor experiência para o usuário.

Conforme os *browsers* evoluem, o *javascript* também precisa evoluir. Uma prova desse processo de crescimento foi a criação do *AJAX*. Um dos pioneiros nesse paradigma foi o *Google*, com o intuito de melhorar a experiência de uso do *Gmail* e evitar que a cada email aberto gerasse uma nova atualização da página; esse tipo de cenário propiciou o começo dos trabalhos para habilitar chamadas *HTTP* a partir do *javascript* e assim evitar a necessidade de atualizar a página para receber conteúdo de um servidor e mostrar na tela. Em 18 de Fevereiro de 2005 *Jesse James Garreth* publicou o artigo *Ajax new approach web applications*[2] disseminando o termo *AJAX* a comunidade, termo esse que revolucionou a maneira de comunicar com servidores até então conhecida.

Com o surgimento da "nuvem", as aplicações precisavam se tornar escaláveis. A arquitetura de *software* teve que se atualizar, as aplicações precisavam tirar o maior proveito de uma única máquina e utilizar o mínimo de recurso possível. Quando surge a necessidade de aumentar recursos, ao invés de fazer upgrade em uma máquina, uma nova máquina com uma nova instância da aplicação seria inicializada, permitindo assim a divisão da carga, dando origem a termos como micro-serviços. Nessa mesma época o *javascript* chegou ao *server side* com o aparecimento do *Node.js*, novamente revolucionando a maneira de desenvolver *software*.

O *Node.js* trouxe todo o poder do *javascript* para o *server side*, tornando-se o principal aliado de grandes empresas como *Uber* e *Netflix*, as quais lidam com milhares de requisições diariamente. A característica de trabalhar de forma assíncrona e ser guiado por eventos possibilitou a criação de aplicações que precisam de conexão em tempo real.

A comunidade *javascript* é vasta e muito colaborativa, diariamente circulam centenas de novas bibliotecas e *frameworks*, tanto para *front-end* quando para *back-end*. Esse dinamismo confunde os desenvolvedores *Node.js*, pois diferentes de outras linguagens consolidadas no *server side* como o *Java* que possui *frameworks* como *SpringMVC* para desenvolvimento *web* e o *Ruby* com o *framework Ruby on Rails*, o *Node.js* possui uma gama gigantesca de *frameworks web* e a maioria deles não mantém uma convenção. A falta de convenções estabelecidas dificulta o caminho dos

[1] http://stackoverflow.com/insights/survey/2017#technology

[2] http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

novos desenvolvedores, pois cada projeto possui uma arquitetura única o que torna complexa a escolha de um padrão e a comparação das vantagens de cada um. Além disso, o *javascript* e o *Node.js* não possuem uma bateria de ferramentas de teste completa, como os pacotes *xUnit* comuns em outras linguagens. Cada projeto possui a sua combinação de ferramentas de testes para os mais determinados cenários, o que contribui com a confusão quando se está começando.

Esse livro tem como objetivo ajudar você a criar aplicações em *Node.js* utilizando os padrões mais reconhecidos da comunidade e seguindo os mais atuais padrões de qualidade para o desenvolvimento de *software*.

O livro guiará você para entender:

- Qual o diferencial do *Node.js* quando comparado a outras linguagens
- Como desenvolver aplicações com *Node.js* utilizando as últimas funcionalidades do *Ecmascript*
- Como construir aplicações modularizadas e desacopladas
- Como Integrar com banco de dados *NoSQL* utilizando *MongoDB*
- Como desenvolver guiado por testes com *TDD*
- Porque testes facilitam o desenvolvimento
- Como testar aplicações em *javascript*
- Como desenhar *APIs* seguindo o padrão *REST*
- Como prover autenticação e segurança de *APIs*

Para que seja possível reproduzir cenários comuns do dia a dia do desenvolvimento será criada, no decorrer do livro, uma *API* que servirá como guia para a introdução dos mais diferentes tópicos. A *API* terá o domínio de uma loja virtual, pois é um caso no qual é possível cobrir os mais diversos cenários do desenvolvimento de aplicações com *Node.js*. Todo o desenvolvimento será guiado por testes, seguindo o padrão conhecido como *TDD* (*test driven development*), introduzindo as mais diferentes e complexas necessidades de testes.

No final desse livro você terá desenvolvido uma aplicação resiliente, seguindo as melhores práticas do desenvolvimento de *software* e com cobertura de testes. Você estará pronto para desenvolver aplicações utilizando *Node.js* e seguindo os princípios do *TDD*.

Introdução ao *Node.js*

Node.js não é uma linguagem de programação nem tampouco um *framework*. A definição mais apropriada seria: um ambiente de *runtime* para *javascript* que roda em cima de uma *engine* conhecida como *Google v8*. O *Node.js* nasceu de uma ideia do *Ryan Dahl* que buscava uma solução para o problema de acompanhar o progresso de *upload* de arquivos sem ter a necessidade de fazer *pooling* no servidor. Em 2009 na *JSConf EU* ele apresentou o *Node.js* e introduziu o *javascript server side* com *I/O* não bloqueante, ganhando assim o interesse da comunidade que começou a contribuir com o projeto desde a versão 0.x.

A primeira versão do *NPM* (*Node Package Manager*), o gerenciador de pacotes oficial do *Node.js*, foi lançada em 2011 permitindo aos desenvolvedores a criação e publicação de suas próprias bibliotecas e ferramentas. O *npm* é tão importante quanto o próprio *Node.js* e desempenha um fator chave para o sucesso do mesmo.

Nessa época não era fácil usar o *Node*. A frequência em que *breaking changes* era incorporadas quase impossibilitava a manutenção dos projetos. O cenário se estabilizou com o lançamento da versão 0.8, que se manteve com baixo número de *breaking changes*. Mesmo com a frequência alta de atualizações a comunidade se manteve ativa, *frameworks* como *Express* e *Socket.IO* já estavam em desenvolvimento desde 2010 e acompanharam, lado a lado, as versões da tecnologia.

O crescimento do *Node.js* foi rápido e teve altos e baixos como a saída do *Ryan Dahl* em 2012 e a separação dos *core commiters* do *Node.js* em 2014 causada pela discordância dos mesmos com a forma como a *Joyent* (empresa na qual Ryan trabalhava antes de sair do projeto) administrava o projeto. Os *core commiters* decidiram fazer um *fork* do projeto e chama-lo de *IO.js* com a intenção de prover releases mais rápidas e acompanhando as melhorias do *Google V8*.

Essa separação trouxe dor de cabeça a comunidade que não sabia qual dos projetos deveria usar. Então, a *Joyent* e outras grandes empresas como *IBM, Paypal e Microsoft* uniram-se para ajudar a comunidade *Node.js* criando a *Node.js Foundation*[3]. A [*Node.js Foundation*] tem como missão uma administração transparente e o encorajamento da participação da comunidade. Com isso, os projetos *Node.js* e *IO.js* fundiram-se e foi lançada a primeira versão estável do *Node.js*, a versão 4.0.

O *Google V8*

O *V8* é uma *engine* criada pela *Google* para ser usada no *browser chrome*. Em 2008 a *Google* tornou o *V8 open source* e passou a chamá-lo de *Chromium project*. Essa mudança possibilitou que a comunidade entendesse a *engine* em sí, além de compreender como o *javascript* é interpretado e compilado pela mesma.

[3] https://nodejs.org/en/foundation/

O *javascript* é uma linguagem interpretada, o que o coloca em desvantagem quando comparado com linguagens compiladas, pois cada linha de código precisa ser interpretada enquanto o código é executado. O *V8* compila o código para linguagem de máquina, além de otimizar drasticamente a execução usando heurísticas, permitindo que a execução seja feita em cima do código compilado e não interpretado.

Entendendo o *Node.js single thread*

A primeira vista o modelo *single thread* parece não fazer sentido, qual seria a vantagem de limitar a execução da aplicação em somente uma *thread*? Linguagens como *Java*, *PHP* e *Ruby* seguem um modelo onde cada nova requisição roda em uma *thread* separada do sistema operacional. Esse modelo é eficiente mas tem um custo de recursos muito alto, nem sempre é necessário todo o recurso computacional aplicado para executar uma nova *thread*. O *Node.js* foi criado para solucionar esse problema, usar programação assíncrona e recursos compartilhados para tirar maior proveito de uma *thread*.

O cenário mais comum é um servidor *web* que recebe milhões de requisições por segundo; Se o servidor iniciar uma nova *thread* para cada requisição vai gerar um alto custo de recursos e cada vez mais será necessário adicionar novos servidores para suportar a demanda. O modelo assíncrono *single thread* consegue processar mais requisições concorrentes do que o exemplo anterior, com um número bem menor de recursos.

Ser *single thread* não significa que o *Node.js* não usa *threads* internamente, para entender mais sobre essa parte devemos primeiro entender o conceito de *I/O* assíncrono não bloqueante.

I/O assíncrono não bloqueante

Trabalhar de forma não bloqueante facilita a execução paralela e o aproveitamento de recursos, essa provavelmente é a característica mais poderosa do *Node.js*, . Para entender melhor vamos pensar em um exemplo comum do dia a dia. Imagine que temos uma função que realiza várias ações, entre elas uma operação matemática, a leitura de um arquivo de disco e em seguida transforma o resultado em uma *String*. Em linguagens bloqueantes, como *PHP* e *Ruby*, cada ação será executada apenas depois que a ação anterior for encerrada. No exemplo citado a ação de transformar a *String* precisa esperar uma ação de ler um arquivo de disco, que pode ser uma operação pesada, certo? Vamos ver um exemplo de forma síncrona, ou seja, bloqueante:

```
const fs = require('fs');
let fileContent;
const someMath = 1+1;

try {
  fileContent = fs.readFileSync('big-file.txt', 'utf-8');
  console.log('file has been read');
} catch (err) {
  console.log(err);
}

const text = `The sum is ${ someMath }`;

console.log(text);
```

Nesse exemplo, a última linha de código com o ***console.log*** precisa esperar a função ***readFileSync*** do módulo de *file system* executar, mesmo não possuindo ligação alguma com o resultado da leitura do arquivo.

Esse é o problema que o *Node.js* se propôs a resolver, possibilitar que ações não dependentes entre sí sejam desbloqueadas. Para solucionar esse problema o *Node.js* depende de uma funcionalidade chamada *high order functions*. As *high order functions* possibilitam passar uma função por parâmetro para outra função, as funções passadas como parâmetro serão executadas posteriormente, como no exemplo a seguir:

```
const fs = require('fs');

const someMath = 1+1;

fs.readFile('big-file.txt', 'utf-8', function (err, content) {
    if (err) {
      return console.log(err)
    }
    console.log(content)
});

const text = `The response is ${ someMath }`;

console.log(text);
```

No exemplo acima usamos a função ***readFile*** do módulo *file system*, assíncrona por padrão. Para que seja possível executar alguma ação quando a função terminar de ler o arquivo é necessário passar uma função por parâmetro, essa função será chamada automaticamente quando a função

readFile finalizar a leitura. Funções passadas por parâmetro para serem chamadas quando a ação é finalizada são chamadas de *callbacks*. No exemplo acima o *callback* recebe dois parâmetros injetados automaticamente pelo *readFile*: *err*, que em caso de erro na execução irá possibilitar o tratamento do erro dentro do *callback*, e *content* que é a resposta da leitura do arquivo.

Para entender como o *Node.js* faz para ter sucesso com o modelo assíncrono é necessário entender também o *Event Loop*.

Event Loop

O *Node.js* é uma linguagem guiada por eventos. O conceito de *Event Driven* é bastante aplicado em interfaces para o usuário, o *javascript* possui diversas *APIs* baseadas em eventos para interações com o *DOM* como por exemplo eventos como *click*, *scroll*, *change* são muito comuns no contexto do *front-end* com *javascript*.

Event driven é um fluxo de controle determinado por eventos ou alterações de estado, a maioria das implementações possuem um *core* (núcleo) que escuta todos os eventos e chama seus respectivos *callbacks* quando eles são lançados (ou tem seu estado alterado), esse é o resumo do *Event Loop* do *Node.js*.

Separadamente, a responsabilidade do *Event Loop* parece simples mas quando nos aprofundamos no funcionamento do *Node.js* notamos que o *Event Loop* é a peça chave para o sucesso do modelo *event driven*. Nos próximos tópicos vamos entender cada um dos componentes que formam o ambiente do *Node.js*, como funcionam e como se conectam.

Call Stack

A *stack* (pilha) é um conceito bem comum no mundo das linguagens de programação, frequentemente se ouve algo do tipo: "Estourou a pilha!". No *Node.js*, e no *javascript* em geral, esse conceito não se difere muito de outras linguagens, sempre que uma função é executada ela entra na *stack*, que executa somente uma coisa por vez, ou seja, o código posterior ao que está rodando precisa esperar a função atual terminar de executar para seguir adiante. Vamos ver um exemplo:

```
1  function generateBornDateFromAge(age) {
2    return 2016 - age;
3  }
4
5  function generateUserDescription(name, surName, age) {
6    const fullName = `${name} ${surName}`;
7    const bornDate = generateBornDateFromAge(age);
8
9    return `${fullName} is ${age} old and was born in ${bornDate}`;
```

```
10  }
11
12  generateUserDescription("Waldemar", "Neto", 26);
```

Para quem já é familiarizado com *javascript* não há nada especial acontecendo aqui. A função ***generateUserDescription*** é chamada recebendo nome, sobrenome e idade de um usuário e retorna uma sentença com as informações colhidas. A função *generateUserDescription* depende da função ***generateBornDateFromAge*** para calcular o ano que o usuário nasceu. Essa dependência será perfeita para entendermos como a *stack* funciona.

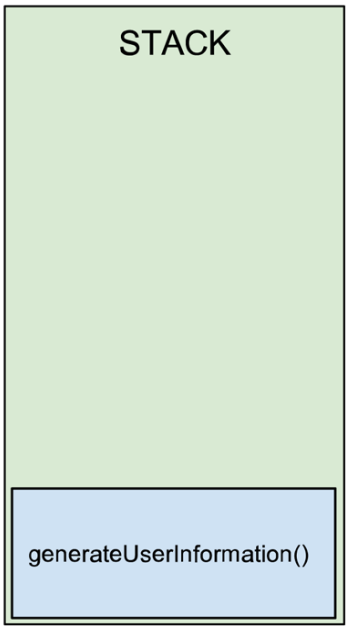
```
function generateBornDateFromAge(age) {
  return 2016 - age;
}

function generateUserInformation(name, surName, age) {
  const fullName = name + " " + surName;
  const bornDate = generateBornDateFromAge(age);

  return fullName + " is " + age + " old and was born in " + bornDate;
}

const userInformation = generateUserInformation("Waldemar", "Neto", 26);

console.log(userInformation);
```



No momento em que a função *generateUserInformation* é invocada ela vai depender da função *generateBornDateFromAge* para descobrir o ano em que o usuário nasceu com base no parâmetro *age*. Quando a função *generateBornDateFromAge* for invocada pela função *generateUserInformation* ela será adicionada a *stack* como no exemplo a seguir:

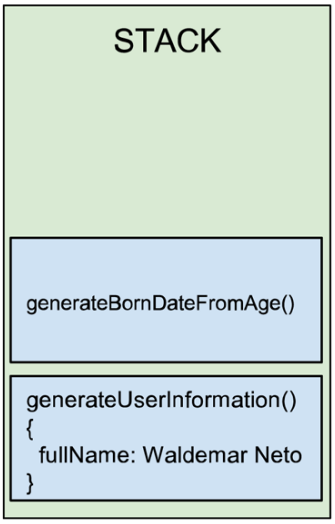
```
function generateBornDateFromAge(age) {
  return 2016 - age;
}

function generateUserInformation(name, surName, age) {
  const fullName = name + " " + surName;
  const bornDate = generateBornDateFromAge(age);

  return fullName + " is " + age + " old and was born in " + bornDate;
}

const userInformation = generateUserInformation("Waldemar", "Neto", 26);

console.log(userInformation);
```



Conforme a função *generateUserInformation* vai sendo interpretada os valores vão sendo atribuídos às respectivas variáveis dentro de seu escopo, como no exemplo anterior. Para atribuir o valor a variável *bornDate* foi necessário invocar a função *generateBornDateFromAge* que quando invocada é imediatamente adicionada a *stack* até que a execução termine e a resposta seja retornada. Após o retorno a *stack* ficará assim:

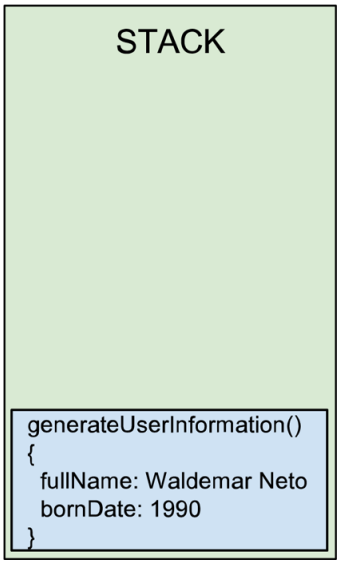
```
function generateBornDateFromAge(age) {
  return 2016 - age;
}

function generateUserInformation(name, surName, age) {
  const fullName = name + " " + surName;
  const bornDate = generateBornDateFromAge(age);

  return fullName + " is " + age + " old and was born in " + bornDate;
}

const userInformation = generateUserInformation("Waldemar", "Neto", 26);

console.log(userInformation);
```



O último passo da função será concatenar as variáveis e criar uma frase, isso não irá adicionar mais nada a *stack*. Quando a função *generateUserInformation* terminar, as demais linhas serão interpretadas. No nosso exemplo o ***console.log*** será executado e vai imprimir o valor da variável *userInformation*.

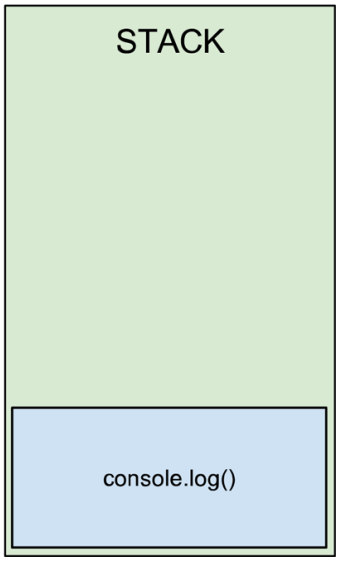
```
function generateBornDateFromAge(age) {
  return 2016 - age;
}

function generateUserInformation(name, surName, age) {
  const fullName = name + " " + surName;
  const bornDate = generateBornDateFromAge(age);

  return fullName + " is " + age + " old and was born in " + bornDate;
}

const userInformation = generateUserInformation("Waldemar", "Neto", 26);

console.log(userInformation);
```



Como a *stack* só executa uma tarefa por vez foi necessário esperar que a função anterior executasse e finalizasse, para que o *console.log* pudesse ser adicionado a *stack*. Entendendo o funcionamento da *stack* podemos concluir que funções que precisam de muito tempo para execução irão ocupar mais tempo na *stack* e assim impedir a chamada das próximas linhas.

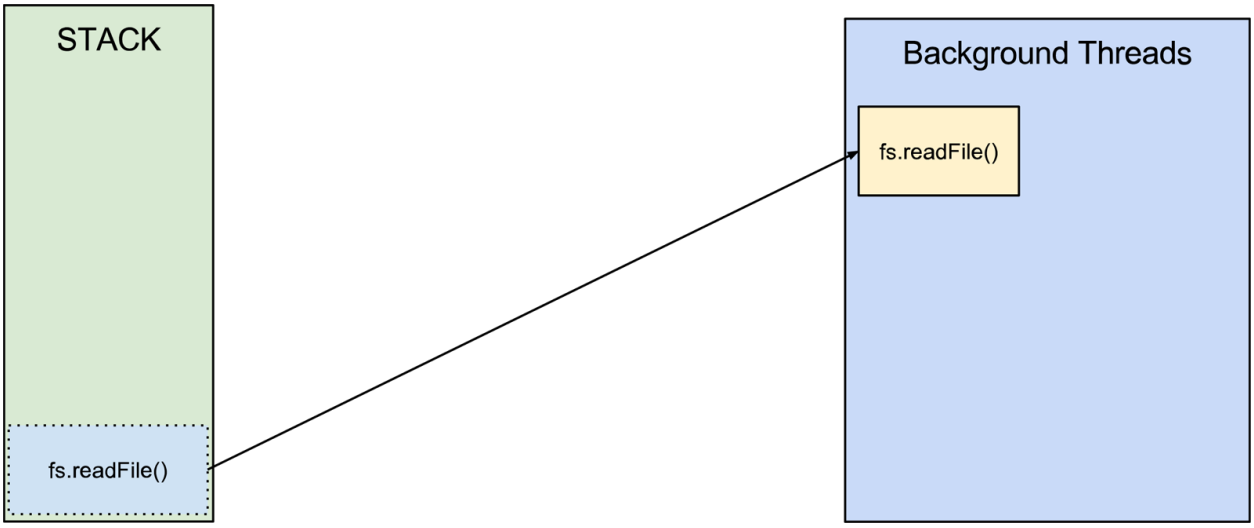
Multithreading

Mas o *Node.js* não é *single thread*? Essa é a pergunta que os desenvolvedores *Node.js* provavelmente mais escutam. Na verdade quem é *single thread* é o *V8*. A *stack* que vimos no capitulo anterior faz parte do *V8*, ou seja, ela é *single thread*. Para que seja possível executar tarefas assíncronas o *Node.js* conta com diversas outras *APIs*, algumas delas providas pelos próprios sistemas operacionais, como é o caso de eventos de disco, *sockets TCP* e *UDP*. Quem toma conta dessa parte de *I/O* assíncrono, de administrar múltiplas *threads* e enviar notificações é a ***libuv***.

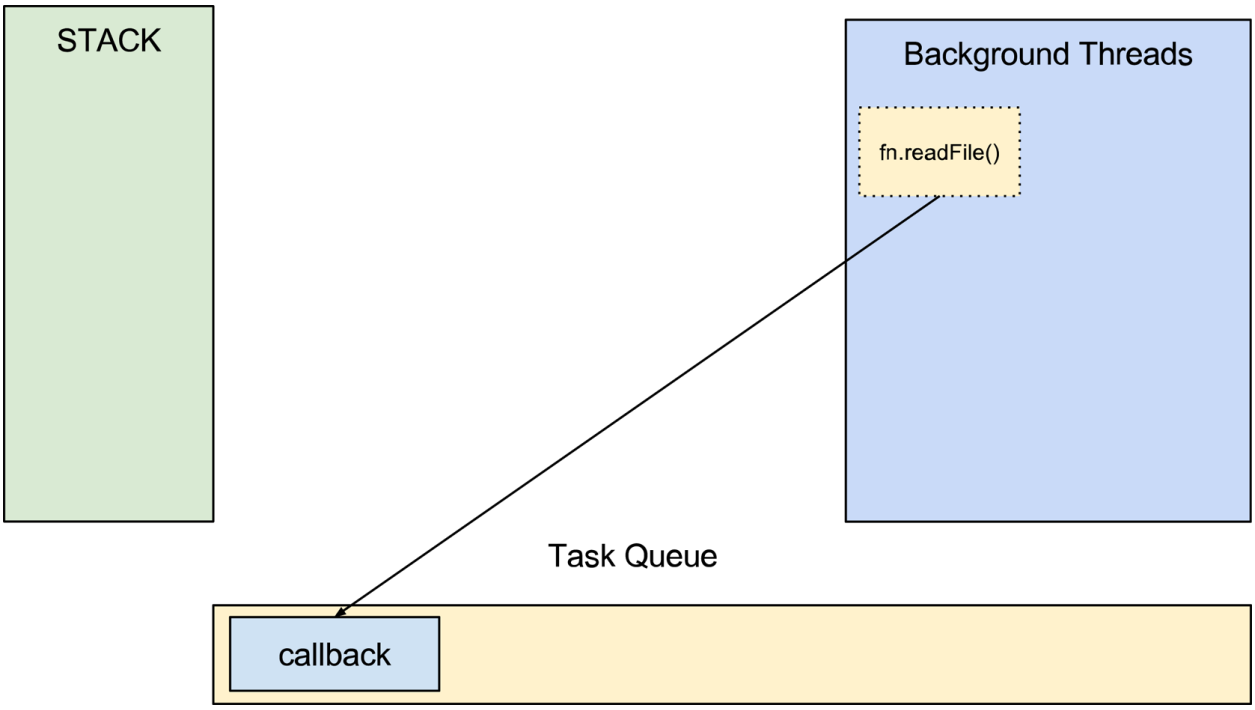
A *libuv*[4] é uma biblioteca *open source* multiplataforma escrita em *C*, criada inicialmente para o *Node.js* e hoje usada por diversos outros projetos como *Julia*[5] e *Luvit*[6].

O exemplo a seguir mostra uma função assíncrona sendo executada:

[4] https://github.com/libuv/libuv
[5] http://julialang.org/
[6] https://luvit.io/



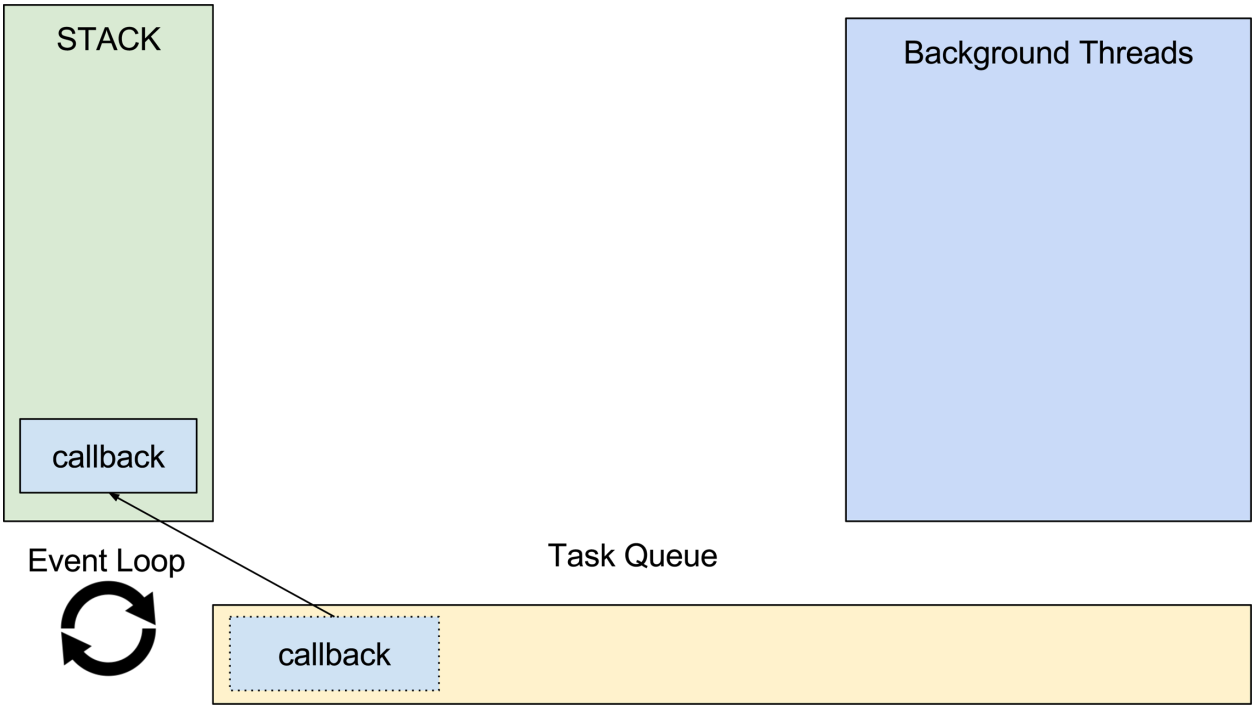
Nesse exemplo a função ***readFile*** do módulo de *file system* do *Node.js* é executada na *stack* e jogada para uma *thread*, a *stack* segue executando as próximas funções enquanto a função *readFile* está sendo administrada pela *libuv* em outra *thread*. Quando ela terminar, o *callback* sera adicionado a uma fila chamada ***Task Queue*** para ser executado pela *stack* assim que ela estiver livre.



Task Queue

Como vimos no capítulo anterior, algumas ações como *I/O* são enviadas para serem executadas em outra *thread* permitindo que o *V8* siga trabalhando e a *stack* siga executando as próximas funções.

Essas funções enviadas para que sejam executadas em outra *thread* precisam de um *callback*. Um *callback* é basicamente uma função que será executada quando a função principal terminar. Esses *callbacks* podem ter responsabilidades diversas, como por exemplo, chamar outras funções e executar alguma lógica. Como o *V8* é *single thread* e só existe uma *stack*, os *callbacks* precisam esperar a sua vez de serem chamados. Enquanto esperam, os *callbacks* ficam em um lugar chamado *task queue* ou fila de tarefas. Sempre que a *thread* principal finalizar uma tarefa, o que significa que a *stack* estará vazia, uma nova tarefa é movida da *task queue* para a *stack* onde será executada. Para entender melhor vamos ver a imagem abaixo:



Esse *loop*, conhecido como ***Event Loop***, é infinito e será responsável por chamar as próximas tarefas da *task queue* enquanto o *Node.js* estiver rodando.

Micro e Macro Tasks

Até aqui vimos como funciona a *stack*, o *multithread* e também como são enfileirados os *callbacks* na *task queue*. Agora vamos conhecer os tipos de *tasks* (tarefas) que são enfileiradas na *task queue*, que podem ser *micro tasks* ou *macro tasks*.

Macro tasks

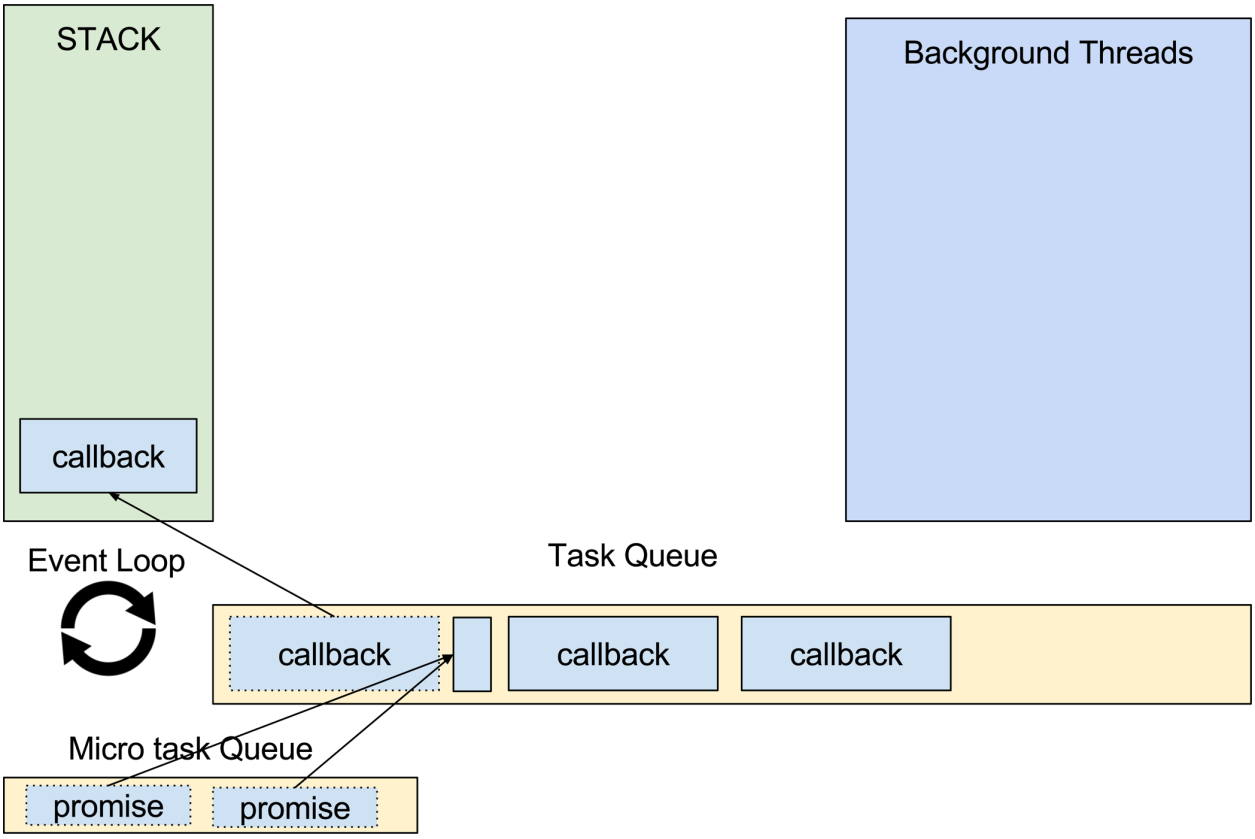
Alguns exemplos conhecidos de *macro tasks* são: *setTimeout*, *I/O*, *setInterval*. Segundo a especificação do *WHATWG*[7] somente uma *macro task* deve ser processada em um ciclo do *Event Loop*.

[7] https://html.spec.whatwg.org/multipage/webappapis.html#task-queue

Micro tasks

Alguns exemplos conhecidos de *micro tasks* são as *promises* e o *process.nextTick*. As *micro tasks* normalmente são tarefas que devem ser executadas rapidamente após alguma ação, ou realizar algo assíncrono sem a necessidade de inserir uma nova *task* na *task queue*. A especificação do *WHATWG* diz que após o *Event Loop* processar a *macro task* da *task queue* todas as *micro tasks* disponíveis devem ser processadas e, caso elas chamem outras *micro tasks*, essas também devem ser resolvidas para que somente então ele chame a próxima *macro task*.

O exemplo abaixo demonstra como funciona esse fluxo:



Configuração do ambiente de desenvolvimento

Durante todo o livro a versão usada do *Node.js* será a 6.9.1 *LTS* (*long term support*). Para que seja possível usar as funcionalidades mais atuais do *javascript* será necessário o *Ecmascript* na versão 6 *ES6* (*ES2015* ou *javascript* 2015), aqui iremos chamar de *ES6*.

Como a versão do *Node.js* que usaremos não dá suporte inteiramente ao *ES6* será necessário o uso de um *transpiler* que vai tornar possível a utilização de 100% das funcionalidades do *ES6*.

O que é um *transpiler*

Transpilers também são conhecidos como compiladores *source-to-source*. Usando um *transpiler* é possível escrever código utilizando as funcionalidade do *ES6* ou versões mais novas e transformar o código final em um código suportado pela versão do *Node.js* que estaremos usando, no caso a 6.x. Um dos *transpilers* mais conhecidos do universo *javascript* é o ***Babel.js***[8]. Criado em 2015 por Sebastian McKenzie, o *Babel* permite utilizar as últimas funcionalidades do *javascript* e ainda assim executar o código em *browser engines* que ainda não as suportam nativamente, como no caso do *v8* (*engine* do *chrome* na qual o *Node.js* roda), pois ele traduz o código gerado para uma forma entendível.

Gerenciamento de projeto e dependências

A maioria das linguagens possuem um gerenciador, tanto para automatizar tarefas, *build*, executar testes quanto para gerenciar dependencias. O *javascript* possui uma variada gama de gerenciadores, como o *Grunt*[9], *Gulp*[10] e *Brocoli*[11] para gerenciar e automatizar tarefas e o *Bower*[12] para gerenciar dependencias de projetos *front-end*. Para o ambiente *Node.js* é necessário um gerenciador que também permita a automatização de tarefas e customização de *scripts*.

Nesse cenário entra o *npm*[13] (*Node Package Manager*), criado por Isaac Z. Schlueter o *npm* foi adotado pelo *Node.js* e é instalado automaticamente junto ao Node. O *npm registry* armazena mais de 400,000 pacotes públicos e privados de milhares de desenvolvedores e empresas possibilitando a divisão e contribuição de pacotes entre a comunidade. O cliente do *npm* (interface de linha de comando) permite utilizar o *npm* para criar projetos, automatizar tarefas e gerenciar dependências.

[8] https://babeljs.io/
[9] https://gruntjs.com/
[10] http://gulpjs.com/
[11] http://broccolijs.com/
[12] https://bower.io/
[13] https://www.npmjs.com/

Iniciando o projeto

Para iniciar um projeto em *Node.js* a primeira coisa a fazer é inicializar o *npm* no diretório onde ficará a aplicação. Para isso, primeiro certifique-se de já ter instalado o *Node.js* e o *npm* em seu computador, caso ainda não os tenha vá até o site do *Node.js* e faça o download https://nodejs.org/en/download/[14]. Ele irá instalar automaticamente também o *npm*.

Configuração inicial

Crie o diretório onde ficará sua aplicação, após isso, dentro do diretório execute o seguinte comando:

```
1  $ npm init
```

Semelhante ao *git*, o *npm* inicializará um novo projeto nesse diretório, depois de executar o comando o *npm* vai apresentar uma série de perguntas (não é necessário respondê-las agora, basta pressionar *enter*. Você poderá editar o arquivo de configuração depois) como:

1. **name**, referente ao nome do projeto.
2. **version**, referente a versão.
3. **description**, referente a descrição do projeto que está sendo criado.
4. **entry point**, arquivo que será o ponto de entrada caso o projeto seja importado por outro.
5. **test command**, comando que executará os testes de aplicação.
6. **git repository**, repositório git do projeto.
7. **keywords**, palavras chave para ajudar outros desenvolvedores a encontrar o seu projeto no *npm*.
8. **author**, autor do projeto.
9. **license** referente a licença de uso do código.

Após isso um arquivo chamado **package.json** será criado com o conteúdo semelhante a:

[14] https://nodejs.org/en/download/

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

O **package.json** é responsável por guardar as configurações *npm* do nosso projeto, nele ficarão nossos *scripts* para executar a aplicação e os testes.

Configurando suporte ao *Ecmascript* 6

Como vimos anteriormente o *Babel* sera responsável por nos permitir usar as funcionalidades do *ES6*, para isso precisamos instalar os pacotes e configurar o nosso ambiente para suportar o *ES6* por padrão em nossa aplicação. O primeiro passo é instalar os pacotes do *Babel*:

```
$ npm install --save-dev babel-cli
```

Após instalar o *Babel* é necessário instalar o *preset* que será usado, no nosso caso será o *ES6*:

```
$ npm install --save-dev babel-preset-node6
```

Note que sempre usamos *–save-dev* para instalar dependências referentes ao *Babel* pois ele não deve ser usado diretamente em produção, para produção vamos compilar o código, veremos isso mais adiante.

O último passo é informar para o *Babel* qual *preset* iremos usar, para isso basta criar um arquivo no diretório raiz da nossa aplicação chamado **.babelrc** com as seguintes configurações:

```
{
  "presets": ["node6"]
}
```

Feito isso a aplicação já estará suportando 100% o *ES6* e será possível utilizar as funcionalidades da versão. O código dessa etapa está disponivel neste link[15].

[15] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step1

Configurando o servidor web

Como iremos desenvolver uma aplicação *web* precisaremos de um servidor que nos ajude a trabalhar com requisições *HTTP*, transporte de dados, rotas e etc. Existem muitas opções no universo *Node.js* como o *Sails.js*[16], *Hapi.js*[17] e *Koa.js*[18]. Vamos optar pelo *Express.js*[19] por possuir um bom tempo de atividade, muito conteúdo na comunidade e é mantido pela *Node Foundation*[20].

O *Express* é um *framework* para desenvolvimento *web* para *Node.js* inspirado no *Sinatra* desenvolvido para o *ruby on rails*. Criado por *TJ Holowaychuk* o *Express* foi adquirido pela *StrongLoop*[21] em 2014 e é administrado atualmente pela *Node.js Foundation*.

Para começar será necessário instalar dois modulos: o *express* e o *body-parser*. Conforme o exemplo a seguir:

```
1 $ npm install --save express body-parser
```

Quando uma requisição do tipo *POST* ou *PUT* é realizada, o corpo da requisição é transportado como texto. Para que seja possível transportar dados como *JSON* (*JavaScript Object Notation*) por exemplo existe o modulo *body-parser*[22] que é um conjunto de *middlewares* para o *express* que analisa o corpo de uma requisição e transforma em algo definido, no nosso caso, em *JSON*.

Agora vamos criar um arquivo chamado **server.js** no diretório raiz e nele vamos fazer a configuração básica do *express*:

```
1  import express from 'express';
2  import bodyParser from 'body-parser';
3
4  const app = express();
5  app.use(bodyParser.json());
6
7  app.get('/', (req, res) => res.send('Hello World!'));
8
9  app.listen(3000, () => {
10   console.log('Example app listening on port 3000!');
11 });
```

[16] http://sailsjs.com/
[17] https://hapijs.com/
[18] http://koajs.com/
[19] https://expressjs.com/
[20] https://nodejs.org/en/foundation/
[21] https://strongloop.com/
[22] https://github.com/expressjs/body-parser

A primeira instrução no exemplo acima é a importação dos módulos *express* e *body-parser* que foram instalados anteriormente. Em seguida uma nova instância do *express* é criada e associada a constante *app*. Para utilizar o *body-parser* é necessário configurar o *express* para utilizar o *middleware*, o *express* possui um método chamado *use* onde é possível passar *middlewares* como parâmetro, no código acima foi passado o **bodyParser.json()** responsável por transformar o corpo das requisições em *JSON*.

A seguir é criada uma rota, os verbos *HTTP* como *GET*, *POST*, *PUT*, *DELETE* são funções no *express* que recebem como parâmetro um padrão de rota, no caso acima /, e uma função de *callback* que será chamada quando a rota receber uma requisição. Os parametros **req** e **res** representam *request* (requisição) e *response* (resposta) e serão injetados automaticamente pelo *express* quando a requisição for recebida. Para finalizar, a função **listen** é chamada recebendo um número referente a porta na qual a aplicação ficará exposta, no nosso caso é a porta 3000.

O último passo é configurar o *package.json* para iniciar nossa aplicação, para isso vamos adicionar um *script* de *start* dentro do objeto *scripts*:

```
"scripts": {
  "start": "babel-node ./server.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Alterado o *package.json* basta executar o comando:

```
$ npm start
```

Agora a aplicação estará disponível em **http://localhost:3000/**. O código dessa etapa está disponivel neste link[23].

Express Middlewares

Middlewares são funções que tem acesso aos objetos: requisição (*request*), resposta (*response*), e o próximo *middleware* que será chamado, normalmente nomeado como *next*. Essas funções são executadas em ordem de chamada, dessa maneira é possível transformar os objetos de requisição e resposta, realizar validações, autenticações e até mesmo terminar a requisição antes que ela execute e lógica escrita na rota. O exemplo a seguir mostra uma aplicação *express* simples com uma rota que devolve um *"Hello world"* quando chamada, nessa rota será adicionado um *middleware*.

[23] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step2

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

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000);
```

Middlewares são apenas funções que recebem os parâmetros requisição (*req*), resposta (*res*) e próximo (*next*), executam alguma lógica e chamam o próximo *middleware*, caso não exista um próximo o *middleware* chama a função da rota. No exemplo abaixo é criado um *middleware* que vai escrever *"LOGGED"* no terminal.

```
const myLogger = function (req, res, next) {
  console.log('LOGGED');
  next();
};
```

Para que o *express* use essa função é necessário passá-la por parâmetro para a função **use**:

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

const myLogger = function (req, res, next) {
  console.log('LOGGED');
  next();
};

app.use(myLogger);

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000);
```

Dessa maneira a cada requisição para qualquer rota o *middleware* será invocado e irá escrever *"LOGGED"* no terminal. *Middlewares* também podem ser invocados em uma rota específica:

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

const myLogger = function (req, res, next) {
  console.log('LOGGED');
  next();
};

app.get('/', myLogger, function (req, res) {
  res.send('Hello World!');
});

app.listen(3000);
```

Esse comportamento é muito útil e colabora com a não duplicação de código, vamos ver mais sobre os *middlewares* no decorrer do livro quando aplicarmos na nossa API.

Desenvolvimento guiado por testes

Agora que vamos começar a desenvolver nossa aplicação precisamos garantir que a responsabilidade, as possíveis rotas, as requisições e as respostas estão sendo atendidas; que estamos entregando o que prometemos e que está tudo funcionando. Para isso, vamos seguir um modelo conhecido como *TDD* (*Test Driven Development* ou Desenvolvimento Guiado por Testes).

TDD - Test Driven Development

O *TDD* é um processo de desenvolvimento de *software* que visa o *feedback* rápido e a garantia de que o comportamento da aplicação está cumprindo o que é requerido. Para isso, o processo funciona em ciclos pequenos e os requerimentos são escritos como casos de teste.

A prática do *TDD* aumentou depois que *Kent Beck* publicou o livro *TDD - Test Driven Development*[24] e fomentou a discussão em torno do tema. Grandes figuras da comunidade ágil como *Martin Fowler* também influenciaram na adoção dessa prática publicando artigos, ministrando palestras e compartilhando *cases* de sucesso.

Os ciclos do *TDD*

Quando desenvolvemos guiados por testes, o teste acaba se tornando uma consequência do processo, ja que vai ser ele que vai determinar o comportamento esperado da implementação. Para que seja possível validar todas as etapas, o *TDD* se divide em ciclos que seguem um padrão conhecido como: ***Red***, ***Green***, ***Refactor***.

Red

Significa escrever o teste antes da funcionalidade e executá-lo. Nesse momento, como a funcionalidade ainda não foi implementada, o teste deve quebrar. Essa fase também serve para verificar se não há erros na sintáxe e na semântica.

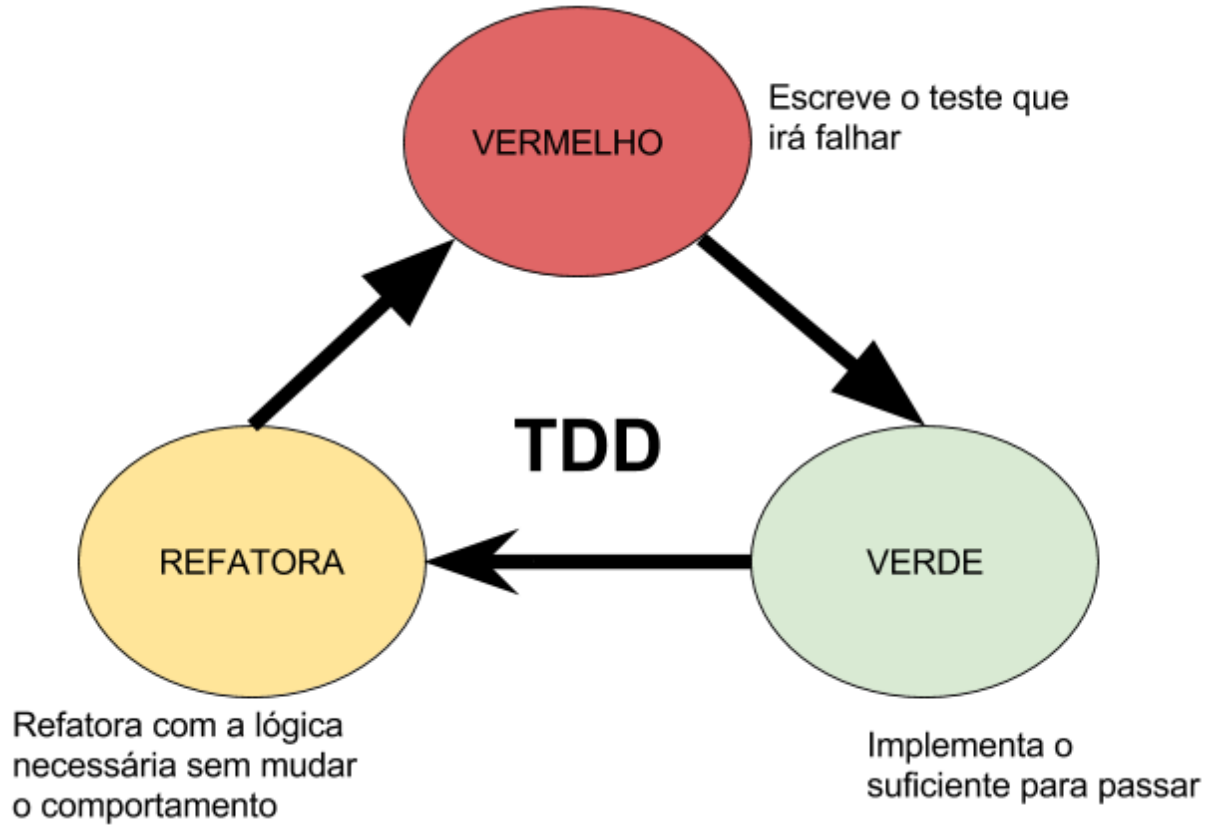
Green

Refere-se a etapa em que a funcionalidade é adicionada para que o teste passe. Nesse momento ainda não é necessário ter a lógica definida mas é importante atender aos requerimentos do teste. Aqui podem ser deixados *to-dos*, dados estáticos, *fixmes*, ou seja, o suficiente para o teste passar.

[24] https://www.amazon.com/Test-Driven-Development-Kent-Beck/dp/0321146530

Refactor

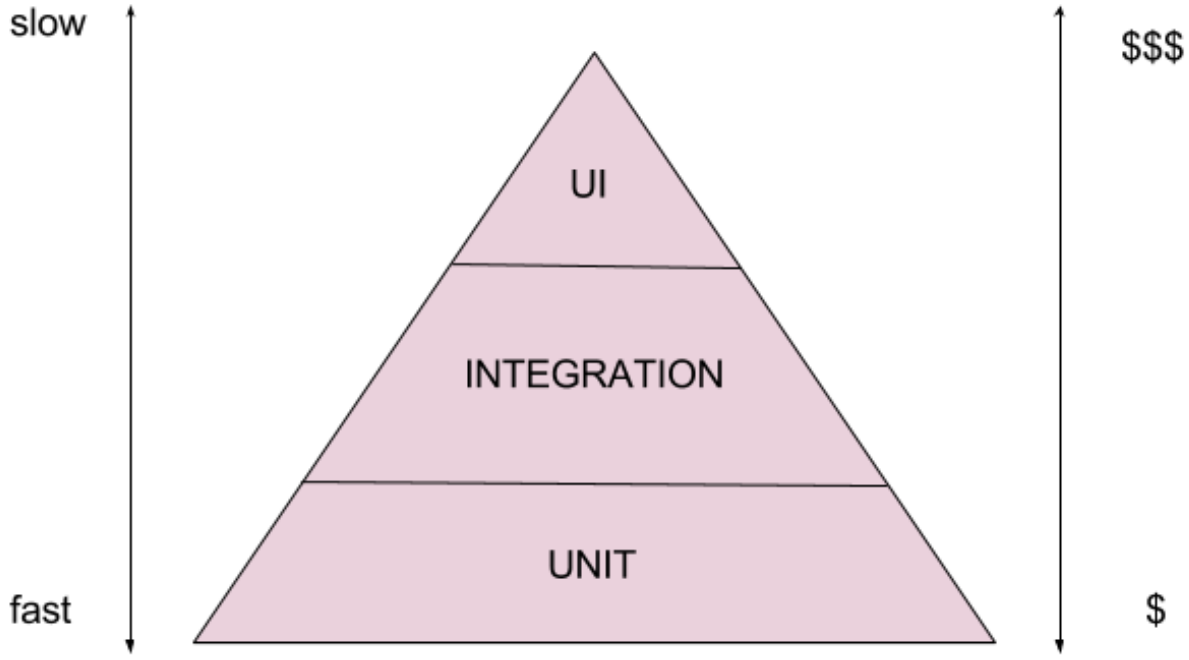
É onde se aplica a lógica necessária. Como o teste já foi validado nos passos anteriores, o *refactor* garantirá que a funcionalidade está sendo implementada corretamente. Nesse momento devem ser removidos os dados estáticos e todos itens adicionadas apenas para forçar o teste a passar, em seguida deve ser feita a implementação real para que o teste volte a passar. A imagem abaixo representa o ciclo do *TDD*:



A pirâmide de testes

A pirâmide de testes é um conceito criado por *Mike Cohn*, escritor do livro *Succeeding with Agile*[25]. O livro propõe que hajam mais testes de baixo nível, ou seja, testes de unidade, depois testes de integração e no topo os testes que envolvem interface.

[25] https://www.amazon.com/Succeeding-Agile-Software-Development-Using/dp/0321579364



O autor observa que os testes de interface são custosos, para alguns testes é necessário inclusive licença de *softwares*. Apesar de valioso, esse tipo de teste necessita da preparação de todo um ambiente para rodar e tende a ocupar muito tempo. O que *Mike* defende é ter a base do desenvolvimento com uma grande cobertura de testes de unidade; no segundo nível, garantir a integração entre os serviços e componentes com testes de integração, sem precisar envolver a interface do usuário. E no topo, possuir testes que envolvam o fluxo completo de interação com a *UI*, para validar todo o fluxo.

Vale lembrar que testes de unidade e integração podem ser feitos em qualquer parte da aplicação, tanto no lado do servidor quanto no lado do cliente, isso elimina a necessidade de ter testes complexos envolvendo todo o fluxo.

Os tipos de testes

Atualmente contamos com uma variada gama de testes, sempre em crescimento de acordo com o surgimento de novas necessidades. Os mais comuns são os teste de unidade e integração, nos quais iremos focar aqui.

Testes de unidade (*Unit tests*)

Testes de unidade são a base da pirâmide de testes. Segundo *Martin Fowler*[26] testes unitários são de baixo nível, com foco em pequenas partes do *software* e tendem a ser mais rapidamente executados quando comparados com outros testes, pois testam partes isoladas.

[26] http://martinfowler.com/bliki/UnitTest.html

Mas o que é uma unidade afinal? Esse conceito é divergente e pode variar de projeto, linguagem, time e paradigma de programação. Linguagens orientadas a objeto tendem a ter classes como uma unidade, já linguagens procedurais ou funcionais consideram normalmente funções como sendo uma unidade. Essa definição é algo muito relativo e depende do contexto e do acordo dos desenvolvedores envolvidos no processo. Nada impede que um grupo de classes relacionadas entre sí ou funções, sejam uma unidade.

No fundo, o que define uma unidade é o comportamento e a facilidade de ser isolada das suas dependências (dependências podem ser classes ou funções que tenham algum tipo de interação com a unidade). Digamos que, por exemplo, decidimos que as nossas unidade serão as classes e estamos testando uma função da classe *Billing* que depende de uma função da classe *Orders*. A imagem abaixo ilustra a dependência:

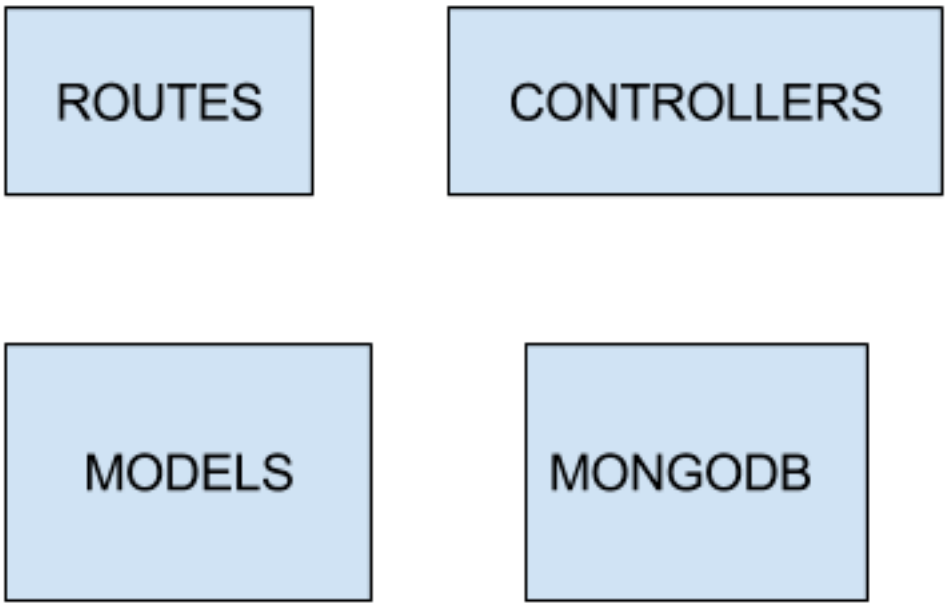
Para testar unitariamente é necessário isolar a classe *Billing* da sua dependência, a classe *Orders*, como na imagem a seguir:

Esse isolamento pode ser feito de diversas maneiras, por exemplo utilizando *mocks* ou *stubs* ou qualquer outra técnica de substituição de dependência e comportamento. O importante é que seja possível isolar a unidade e ter o comportamento esperado da dependência.

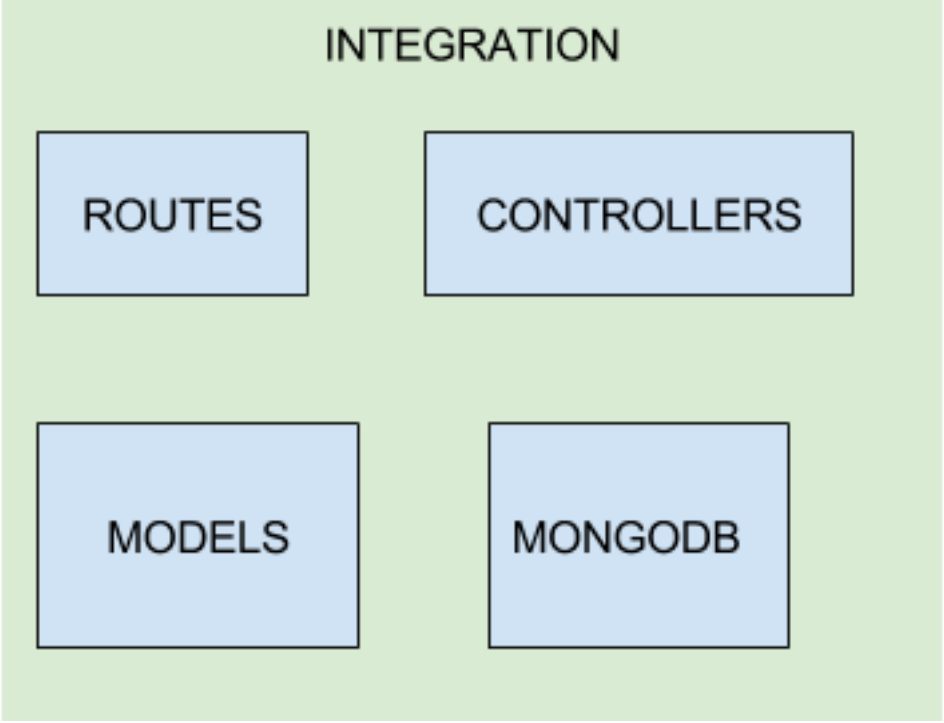
Testes de integração (*Integration tests*)

Testes de integração servem para verificar se a comunicação entre os componentes de um sistema está ocorrendo conforme o esperado. Diferente dos testes de unidade, onde a unidade é isolada de duas dependências, no teste de integração deve ser testado o comportamento da interação entre as unidades. Não há um nível de granularidade específico, a integração pode ser testada em qualquer nível, seja a interação entre camadas, classes ou até mesmo serviços.

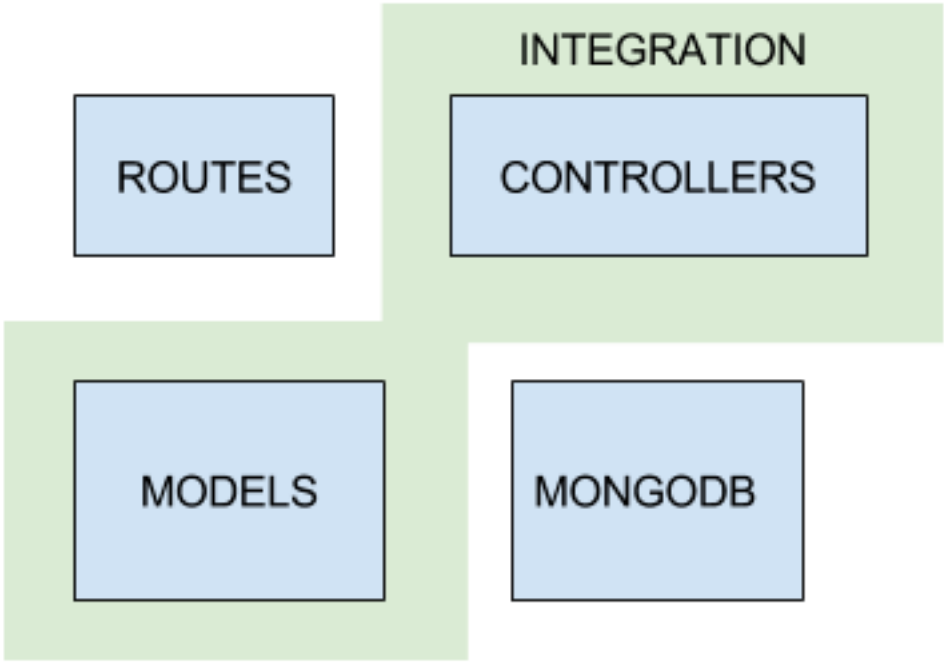
No exemplo a seguir temos uma arquitetura comum de aplicações *Node.js* e desejamos testar a integração entre as rotas, *controllers*, *models* e banco de dados:



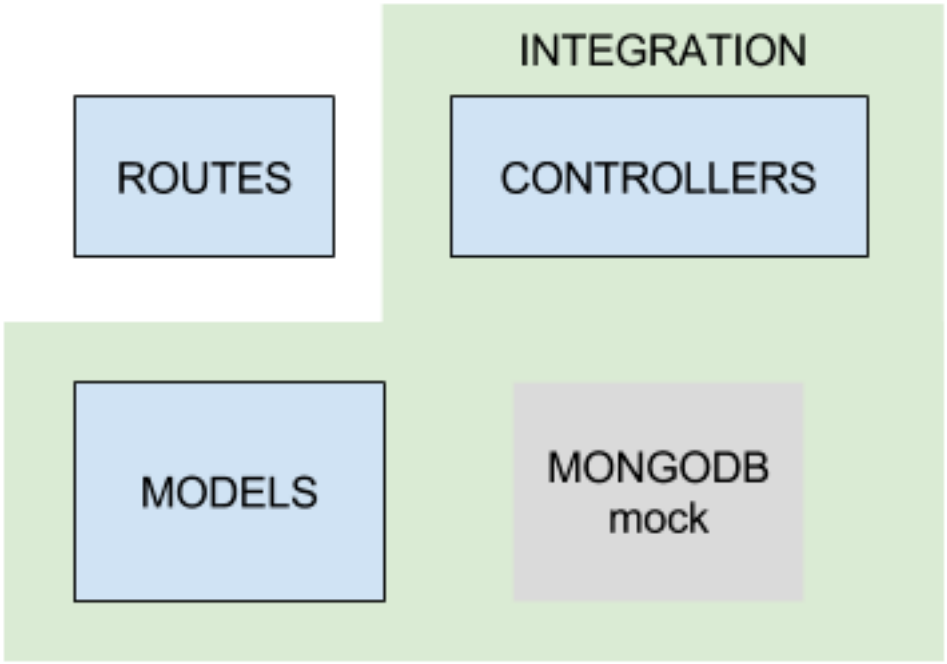
Nossa integração pode ser desde a rota até salvar no banco de dados (nesse caso, *MongoDB*), dessa maneira é possível validar todo o fluxo até o dado ser salvo no banco, como na imagem a seguir:



Esse teste é custoso porém imprescindível. Será necessário limpar o banco de dados a cada teste e criar os dados novamente, além de custar tempo e depender de um serviço externo como o *MongoDB*. Um grau de interação desse nível terá vários possíveis casos de teste, como por exemplo: o usuário mandou um dado errado e deve receber um erro de validação. Para esse tipo de cenário pode ser melhor diminuir a granularidade do teste para que seja possível ter mais casos de teste. Para um caso onde o *controller* chama o *model* passando dados inválidos e a válidação deve emitir um erro, poderíamos testar a integração entre o *controller* e o *model*, como no exemplo a seguir:



Nesse exemplo todos os componentes do sistema são facilmente desacopláveis, podem haver casos onde o *model* depende diretamente do banco de dados e como queremos apenas testar a validação não precisamos inserir nada no banco, nesse caso é possível substituir o banco de dados ou qualquer outra dependência por um *mock* ou *stub* para reproduzir o comportamento de um banco de dados sem realmente chamar o banco.

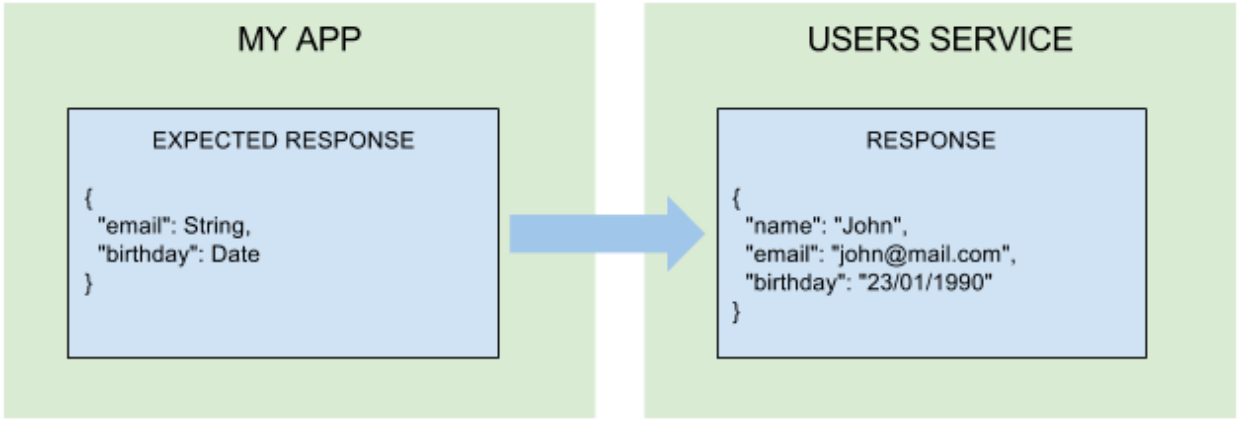


Teste de integração de contrato (*Integration contract tests*)

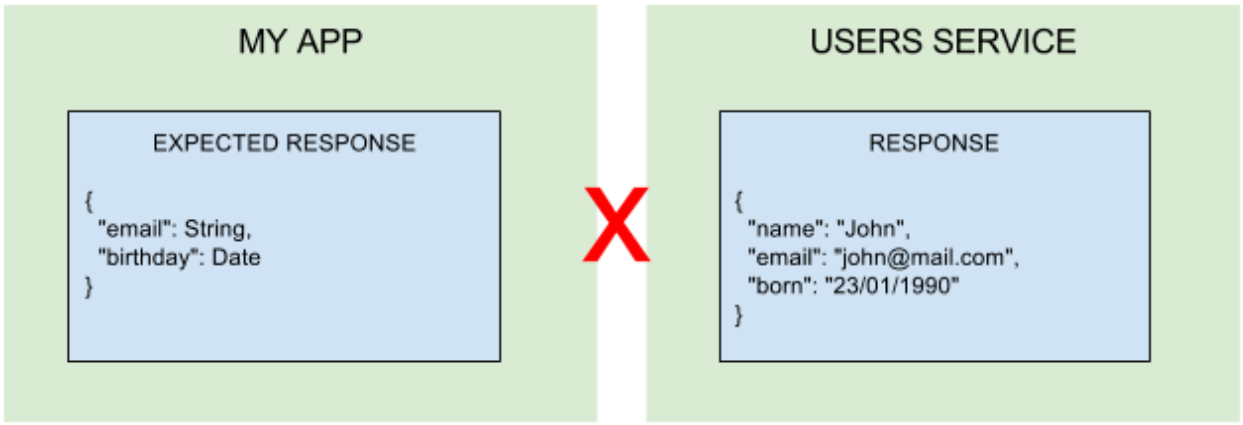
Testes de contrato ganharam muita força devido ao crescimento das *APIs* e dos micro serviços. Normalmente, quando testamos a nossa aplicação, mesmo com o teste de integração, tendemos a não usar os serviços externos e sim um substituto que devolve a resposta esperada. Isso por que serviços externos podem afetar no tempo de resposta da requisição, podem cair, aumentar o custo e isso pode afetar nossos testes. Mas por outro lado, quando isolamos nossa aplicação dos outros serviços para testar ficamos sem garantia de que esses serviços não mudaram suas *APIs* e que a resposta esperada ainda é a mesma. Para solucionar esses problemas existem os testes de contrato.

A definição de um contrato

Sempre que consumimos um serviço externo dependemos de alguma parte dele ou de todos os dados que ele provém e o serviço se compromete a entregar esses dados. O exemplo abaixo mostra um teste de contrato entre a aplicação e um serviço externo, nele é verificado se o contrato entre os dois ainda se mantém o mesmo.



É importante notar que o contrato varia de acordo com a necessidade, nesse exemplo a nossa aplicação depende apenas dos campos *email* e *birthday* então o contrato formado entre eles verifica apenas isso. Se o *name* mudar ele não quebrará nossa aplicação nem o contrato que foi firmado. Em testes de contrato o importante é o tipo e não o valor. No exemplo verificamos se o *email* ainda é *String* e se o campo *birthday* ainda é do tipo *Date*, dessa maneira garantimos que a nossa aplicação não vai quebrar. O exemplo a seguir mostra um contrato quebrado onde o campo *birthday* virou *born*, ou seja, o serviço externo mudou o nome do campo, nesse momento o contrato deve quebrar.



Testes de contrato possuem diversas extensões, o caso acima é chamado de *consumer contract*[27] onde o consumidor verifica o contrato e, caso o teste falhe, notifica o *provider* (provedor) ou altera sua aplicação para o novo contrato. Também existe o *provider contracts* onde o próprio provedor testa se as alterações feitas irão quebrar os consumidores.

Test Doubles

Testar código com *ajax*, *network*, *timeouts*, banco de dados e outras dependências que produzem efeitos colaterais é sempre complicado. Por exemplo, quando se usa *ajax*, ou qualquer outro tipo de

[27] https://www.thoughtworks.com/pt/radar/techniques/consumer-driven-contract-testing

networking, é necessário comunicar com um servidor que irá responder para a requisição; já com o banco de dados será necessário inicializar um serviço para tornar possível o teste da aplicação: limpar e criar tabelas para executar os testes e etc.

Quando as unidades que estão sendo testadas possuem dependências que produzem efeitos colaterais, como os exemplos acima, não temos garantia de que a unidade está sendo testada isoladamente. Isso abre espaço para que o teste quebre por motivos não vinculados a unidade em sí, como por exemplo o serviço de banco não estar disponível ou uma *API* externa retornar uma resposta diferente da esperada no teste.

Há alguns anos atrás Gerard Meszaros publicou o livro *XUnit Test Patterns: Refactoring Test Code* e introduziu o termo *Test Double* (traduzido como "dublê de testes") que nomeia as diferentes maneiras de substituir dependências. A seguir vamos conhecer os mais comuns *test doubles* e quais são suas características, prós e contras.

Para facilitar a explicação será utilizado o mesmo exemplo para os diferentes tipos de *test doubles*, também será usada uma biblioteca de suporte chamada *Sinon.js*[28] que possibilita a utilização de *stubs*, *mocks* e *spies*.

A *controller* abaixo é uma classe que recebe um banco de dados como dependência no construtor. O método que iremos testar unitariamente dessa classe é o método ***getAll***, ele retorna uma consulta do banco de dados com uma lista de usuários.

```
1  const Database = {
2    findAll() {}
3  }
4
5  class UsersController {
6    constructor(Database) {
7      this.Database = Database;
8    }
9
10   getAll() {
11     return this.Database.findAll('users');
12   }
13 }
```

Fake

Durante o teste, é frequente a necessidade de substituir uma dependência para que ela retorne algo específico, independente de como for chamada, com quais parâmetros, quantas vezes, a resposta sempre deve ser a mesma. Nesse momento a melhor escolha são os *Fakes*. *Fakes* podem ser classes, objetos ou funções que possuem uma resposta fixa independente da maneira que forem chamadas. O exemplo abaixo mostra como testar a classe *UsersController* usando um *fake*:

[28] http://sinonjs.org/

```
describe('UsersController getAll()', () => {
  it('should return a list of users', () => {
    const expectedDatabaseResponse = [{
      id: 1,
      name: 'John Doe',
      email: 'john@mail.com'
    }];

    const fakeDatabase = {
      findAll() {
        return expectedDatabaseResponse;
      }
    }
    const usersController = new UsersController(fakeDatabase);
    const response = usersController.getAll();

    expect(response).to.be.eql(expectedDatabaseResponse);
  });
});
```

Nesse caso de teste não é necessária nenhuma biblioteca de suporte, tudo é feito apenas criando um objeto *fake* para substituir a dependência do banco de dados. O método ***findAll*** passa a ter uma resposta fixa, que é uma lista com um usuário. Para validar o teste é necessário verificar se a resposta do método *getAll* do *controller* responde com uma lista igual a declarada no *expectedDatabaseResponse*.

Vantagens:

- Simples de escrever
- Não necessita de bibliotecas de suporte
- Desacoplado da dependencia original

Desvantagens:

- Não possibilita testar múltiplos casos
- Só é possível testar se a saída está como esperado, não é possível validar o comportamento interno da unidade

Quando usar *fakes*:

Fakes devem ser usados para testar dependências que não possuem muitos comportamentos.

Spy

Como vimos anteriormente os *fakes* permitem substituir uma dependência por algo customizado mas não possibilitam saber, por exemplo, quantas vezes uma função foi chamada, quais parâmetros ela recebeu e etc. Para isso existem os *spies*, como o próprio nome já diz, eles gravam informações sobre o comportamento do que está sendo "espionado". No exemplo abaixo é adicionado um *spy* no método *findAll* do *Database* para verificar se ele está sendo chamado com os parâmetros corretos:

```
1  describe('UsersController getAll()', () => {
2    it('should database findAll with correct parameters', () => {
3      const findAll = sinon.spy(Database, 'findAll');
4
5      const usersController = new UsersController(Database);
6      usersController.getAll();
7
8      sinon.assert.calledWith(findAll, 'users');
9      findAll.restore();
10   });
11 });
```

Note que é adicionado um *spy* na função ***findAll*** do *Database*, dessa maneira o *Sinon* devolve uma referência a essa função e também adiciona alguns comportamentos a ela que possibilitam realizar checagens como *sinon.assert.calledWith(findAll, 'users')* onde é verificado se a função foi chamada com o parâmetro esperado.

Vantagens:

- Permite melhor assertividade no teste
- Permite verificar comportamentos internos
- Permite integração com dependências reais

Desvantagens:

- Não permitem alterar o comportamento de uma dependência
- Não é possível verificar múltiplos comportamentos ao mesmo tempo

Quando usar *spies*:

Spies podem ser usados sempre que for necessário ter assertividade de uma dependência real ou, como em nosso caso, em um *fake*. Para casos onde é necessário ter muitos comportamos é provável que *stubs* e *mocks* venham melhor a calhar.

Stub

Fakes e *spies* são simples e substituem uma dependência real com facilidade, como visto anteriormente, porém, quando é necessário representar mais de um cenário para a mesma dependência eles podem não dar conta. Para esse cenário entram na jogada os *Stubs*. *Stubs* são *spies* que conseguem mudar o comportamento dependendo da maneira em que forem chamados, veja o exemplo abaixo:

```
describe('UsersController getAll()', () => {
  it('should return a list of users', () => {
    const expectedDatabaseResponse = [{
      id: 1,
      name: 'John Doe',
      email: 'john@mail.com'
    }];

    const findAll = sinon.stub(Database, 'findAll');
    findAll.withArgs('users').returns(expectedDatabaseResponse);

    const usersController = new UsersController(Database);
    const response = usersController.getAll();

    sinon.assert.calledWith(findAll, 'users');
    expect(response).to.be.eql(expectedDatabaseResponse);
    findAll.restore();
  });
});
```

Quando usamos *stubs* podemos descrever o comportamento esperado, como nessa parte do código:

```
findAll.withArgs('users').returns(expectedDatabaseResponse);
```

Quando a função *findAll* for chamada com o parâmetro *users*, retorna a resposta padrão.

Com *stubs* é possível ter vários comportamentos para a mesma função com base nos parâmetros que são passados, essa é uma das maiores diferenças entre *stubs* e *spies*.

Como dito anteriormente, *stubs* são *spies* que conseguem alterar o comportamento. É possível notar isso na asserção *sinon.assert.calledWith(findAll, 'users')* ela é a mesma asserção do *spy* anterior. Nesse teste são feitas duas asserções, isso é feito apenas para mostrar a semelhança com *spies*, múltiplas asserções em um mesmo caso de teste é considerado uma má prática.

Vantagens:

- Comportamento isolado

- Diversos comportamentos para uma mesma função
- Bom para testar código assíncrono

Desvantagens:

- Assim como spies não é possível fazer múltiplas verificações de comportamento

Quando usar *stubs*:

Stubs são perfeitos para utilizar quando a unidade tem uma dependência complexa, que possui múltiplos comportamentos. Além de serem totalmente isolados os *stubs* também tem o comportamento de *spies* o que permite verificar os mais diferentes tipos de comportamento.

Mock

Mocks e *stubs* são comumente confundidos pois ambos conseguem alterar comportamento e também armazenar informações. *Mocks* também podem ofuscar a necessidade de usar *stubs* pois eles podem fazer tudo que *stubs* fazem. O ponto de grande diferença entre *mocks* e *stubs* é sua responsabilidade: *stubs* tem a responsabilidade de se comportar de uma maneira que possibilite testar diversos caminhos do código, como por exemplo uma resposta de uma requisição *http* ou uma exceção; Já os *mocks* substituem uma dependência permitindo a verificação de múltiplos comportamentos ao mesmo tempo. O exemplo a seguir mostra a classe *UsersController* sendo testada utilizando *Mock*:

```
1  describe('UsersController getAll()', () => {
2    it('should call database with correct arguments', () => {
3      const databaseMock = sinon.mock(Database);
4      databaseMock.expects('findAll').once().withArgs('users');
5
6      const usersController = new UsersController(Database);
7      usersController.getAll();
8
9      databaseMock.verify();
10     databaseMock.restore();
11   });
12 });
```

A primeira coisa a se notar no código é a maneira de fazer asserções com *Mocks*, elas são descritas nessa parte:

```
1  databaseMock.expects('findAll').once().withArgs('users');
```

Nela são feitas duas asserções, a primeira para verificar se o método *findAll* foi chamado uma vez e na segunda se ele foi chamado com o argumento *users*, em seguida o código é executado e é chamada a função *verify()* do *Mock* que irá verificar se as expectativas foram atingidas.

Vantagens:

- Verificação interna de comportamento
- Diversas asserções ao mesmo tempo

Desvantagens:

- Diversas asserções ao mesmo tempo podem tornar o teste difícil de entender.

Quando usar *mocks*:

Mocks são úteis quando é necessário verificar múltiplos comportamentos de uma dependência. Isso também pode ser sinal de um *design* de código mal pensado, onde a unidade tem muita responsabilidade. É necessário ter muito cuidado ao usar *Mocks* já que eles podem tornar os testes pouco legíveis.

O ambiente de testes em *javascript*

Diferente de muitas linguagens que contam com ferramentas de teste de forma nativa ou possuem algum *xUnit (JUnit, PHPUnit, etc)* no *javascript* temos todos os componentes das suites de testes separados, o que nos permite escolher a melhor combinação para a nossa necessidade (mas também pode criar confusão). Em primeiro lugar precisamos conhecer os componentes que fazem parte de uma suíte de testes em *javascript*:

Test runners

Test runners são responsáveis por importar os arquivos de testes e executar os casos de teste. Eles esperam que cada caso de teste devolva *true* ou *false*. Alguns dos test runners mais conhecidos de *javascript* são o *Mocha*[29] e o *Karma*[30].

Bibliotecas de *Assert*

Alguns *test runners* possuem bibliotecas de *assert* por padrão, mas é bem comum usar uma externa. Bibliotecas de *assert* verificam se o teste está cumprindo com o determinado fazendo a afirmação e respondendo com *true* ou *false* para o *runner*. Algumas das bibliotecas mais conhecidas são o *chai*[31] e o *assert*[32].

[29] https://mochajs.org/
[30] https://karma-runner.github.io/1.0/index.html
[31] http://chaijs.com/
[32] https://nodejs.org/api/assert.html

Bibliotecas de suporte

Somente executar os arquivos de teste e fazer o *assert* nem sempre é o suficiente. Pode ser necessário substituir dependências, subir servidores *fake*, alterar o *DOM* e etc. Para isso existem as bibliotecas de suporte. As bibliotecas de suporte se separam em diversas responsabilidades, como por exemplo: para fazer *mocks* e *spys* temos o *SinonJS*[33] e o *TestDoubleJS*[34] já para emular servidores existe o *supertest*[35].

[33] http://sinonjs.org/

[34] https://github.com/testdouble/testdouble.js

[35] https://github.com/visionmedia/supertest

Configurando testes de integração

Iremos testar de fora para dentro, ou seja, começaremos pelos testes de integração e seguiremos para os testes de unidade.

Instalando *Mocha, Chai* e *Supertest*

Para começar vamos instalar as ferramentas de testes com o comando abaixo:

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

Vamos instalar três módulos:

- *Mocha*: módulo que ira executar as suites de teste.
- *Chai*: módulo usado para fazer asserções.
- *Supertest*: módulo usado para emular e abstrair requisições *http*.

Separando execução de configuração

Na sequência, será necessário alterar a estrutura de diretórios da nossa aplicação atual, criando um diretório chamado ***src***, onde ficará o código fonte. Dentro dele iremos criar um arquivo chamado ***app.js*** que terá a responsabilidade de iniciar o *express* e carregar os *middlewares*. Ele ficará assim:

```
import express from 'express';
import bodyParser from 'body-parser';

const app = express();
app.use(bodyParser.json());

app.get('/', (req, res) => res.send('Hello World!'));

export default app;
```

Aqui copiamos o código do *server.js* e removemos a parte do *app.listen*, a qual iniciava a aplicação, e adicionamos o ***export default app*** para exportar o *app* como um módulo.
Agora precisamos alterar o *server.js* no diretório raiz para que ele utilize o *app.js*:

```
import app from './src/app';
const port = 3000;

app.listen(port, () => {
  console.log(`app running on port ${port}`);
});
```

Note que agora separamos a responsabilidade de inicializar o *express* e carregar os *middlewares* da parte de iniciar a aplicação em sí. Como nos testes a aplicação será inicializada pelo *supertest* e não pelo *express* como é feito no *server.js*, esse separação torna isso fácil.

Configurando os testes

Agora que a aplicação está pronta para ser testada, vamos configurar os testes. O primeiro passo é criar o diretório ***test*** na raiz do projeto, e dentro dele o diretório onde ficarão os testes de integração, vamos chamar esse diretório de ***integration***.
A estrutura de diretórios ficará assim:

```
├── package.json
├── server.js
├── src
|   └── app.js
└── test
|   └── integration
```

Dentro de *integration* vamos criar os arquivos de configuração para os testes de integração. O primeiro será referente as configurações do *Mocha*, vamos criar um arquivo chamado ***mocha.opts*** dentro do diretório *integration* com o seguinte código:

```
--require test/integration/helpers.js
--reporter spec
--compilers js:babel-core/register
--slow 5000
```

O primeiro *require* será o arquivo referente as configurações de suporte para os testes, o qual criaremos a seguir. Na linha seguinte definimos qual será o *reporter*, nesse caso, o *spec*[36]. *Reporters* definem o estilo da saída do teste no terminal.
Na terceira linha definimos os *compilers*, como iremos usar *ES6*, também nos testes usaremos o *compiler* do *babel* no *Mocha*. E na última linha o *slow* referente a demora máxima que um caso de teste pode levar, como testes de integração tendem a depender de agentes externos como banco de dados e etc, é necessário ter um tempo maior de *slow* para eles.

O próximo arquivo que iremos criar nesse mesmo diretório é o ***helpers.js***. Ele terá o seguinte código:

[36] https://mochajs.org/#spec

```
import supertest from 'supertest';
import chai from 'chai';
import app from '../../src/app.js';

global.app = app;
global.request = supertest(app);
global.expect = chai.expect;
```

O arquivo *helpers* é responsável por inicializar as configurações de testes que serão usadas em todos os testes de integração, removendo a necessidade de ter de realizar configurações em cada cenário de teste.

Primeiro importamos os módulos necessários para executar os testes de integração que são o *supertest* e o *chai* e também a nossa aplicação *express* que chamamos de *app*.
Depois definimos as globais usando **global**. Globais fazem parte do *Mocha*, tudo que for definido como global poderá ser acessado em qualquer teste sem a necessidade de ser importado.
No nosso arquivo *helpers* configuramos o *app* para ser global, ou seja, caso seja necessário usá-lo em um caso de teste basta chamá-lo diretamente. Também é definido um global chamado ***request***, que é o *supertest* recebendo o *express* por parâmetro.
Lembram que falei da vantagem de separar a execução da aplicação da configuração do *express*? Agora o *express* pode ser executado por um emulador como o *supertest*.
E por último o *expect* do *Chai* que será utilizado para fazer as asserções nos casos de teste.

Criando o primeiro caso de teste

Com as configurações finalizadas agora nos resta criar nosso primeiro caso de teste. Vamos criar um diretório chamado ***routes*** dentro do diretório *integration* e nele vamos criar o arquivo ***products_spec.js*** que vai receber o teste referente as rotas do recurso *products* da nossa *API*.
A estrutura de diretórios deve estar assim:

```
├── package.json
├── server.js
├── src
|   └── app.js
└── test
|  └── integration
|    ├── helpers.js
|    ├── mocha.opts
|    └── routes
|      └── products_spec.js
```

Agora precisamos escrever nosso caso de teste, vamos começar com o seguinte código no arquivo ***products_spec.js***:

```
describe('Routes: Products', () => {

});
```

O ***describe*** é uma global do *Mocha* usada para descrever suítes de testes que contém um ou mais casos de testes e/ou contém outras suítes de testes. Como esse é o *describe* que irá englobar todos os testes desse arquivo seu texto descreve a responsabilidade geral da suíte de testes que é testar a rota *products*.
Agora vamos adicionar um produto padrão para os nossos testes:

```
describe('Routes: Products', () => {
  const defaultProduct = {
    name: 'Default product',
    description: 'product description',
    price: 100
  };
});
```

Como a maioria dos testes precisará de um produto, tanto para inserir quanto para verificar nas buscas, criamos uma constante chamada ***defaultProduct*** para ser reusada pelos casos de teste.
O próximo passo é descrever a nossa primeira suíte de testes:

```
describe('Routes: Products', () => {
  const defaultProduct = {
    name: 'Default product',
    description: 'product description',
    price: 100
  };

  describe('GET /products', () => {
    it('should return a list of products', done => {

    });
  });
});
```

Adicionamos mais um *describe* para deixar claro que todas as suítes de teste dentro dele fazem parte do método *http GET* na rota */products*. Isso facilita a legibilidade do teste e deixa a saída do terminal mais clara.

A função ***it*** também é uma global do *Mocha* e é responsável por descrever um caso de teste.
Descrições de casos de teste seguem um padrão declarativo, como no exemplo acima: *"Isso deve*

retornar uma lista de produtos".
Note que também é passado um parâmetro chamado *done* para o caso de teste, isso ocorre porque testes que executam funções assíncronas, como requisições *http*, precisam informar ao *Mocha* quando o teste finalizou e fazem isso chamando a função *done*.
Vejamos na implementação a seguir:

```
describe('Routes: Products', () => {
  const defaultProduct = {
    name: 'Default product',
    description: 'product description',
    price: 100
  };

  describe('GET /products', () => {
    it('should return a list of products', done => {

      request
      .get('/products')
      .end((err, res) => {
        expect(res.body[0]).to.eql(defaultProduct);
        done(err);
      });
    });
  });
});
```

Na implementação do teste usamos o *supertest* que exportamos globalmente como ***request*** no *helpers.js*. O *supertest* nos permite fazer uma requisição *http* para uma determinada rota e verificar a sua resposta.
Quando a requisição terminar a função *end* será chamada pelo *supertest* e vai receber a resposta ou um erro, caso ocorra. No exemplo acima é verificado se o primeiro elemento da lista de produtos retornada é igual ao nosso *defaultProduct*.
O *expect* usado para fazer a asserção faz parte do *Chai* e foi exposto globalmente no *helpers.js*.
Para finalizar, notificamos o *Mocha* que o teste finalizou chamando a função *done* que recebe *err* como parâmetro, caso algum erro ocorra ele irá mostrar a mensagem de erro no terminal.

Executando os testes

Escrito nosso teste, vamos executá-lo. Para automatizar a execução vamos adicionar a seguinte linha no ***package.json*** dentro de *scripts*:

```
"test:integration": "NODE_ENV=test mocha --opts test/integration/mocha.opts test\
/integration/**/*_spec.js"
```

Estamos adicionando uma variável de ambiente como *test*, que além de boa prática também nos será útil em seguida, e na sequência as configurações do *Mocha*.

Para executar os testes agora basta executar o seguinte comando no terminal, dentro do diretório *root* da aplicação:

```
$ npm run test:integration
```

A saída deve ser a seguinte:

```
Routes: Products
GET /products
1) should return a list of products

0 passing (172ms)
  1 failing


  1) Routes: Products GET /products should return a list of products:
  Uncaught AssertionError: expected undefined to deeply equal { Object (name, de\
scription, ...) }
```

Isso quer dizer que o teste está implementado corretamente, sem erros de sintaxe por exemplo, mas está falhando pois ainda não temos esse comportamento na aplicação. Essa é a etapa ***RED*** do *TDD*, conforme vimos anteriormente.

Fazendo os testes passarem

Escrevemos nossos testes e eles estão no estado ***RED***, ou seja, implementados mas não estão passando. O próximo passo, seguindo o *TDD*, é o ***GREEN*** onde vamos implementar o mínimo para fazer o teste passar.
Para isso, precisamos implementar uma rota na nossa aplicação que suporte o método *http GET* e retorne uma lista com, no mínimo, um produto igual ao nosso *defaultProduct* do teste. Vamos alterar o arquivo *app.js* e adicionar a seguinte rota:

```
app.get('/products', (req, res) => res.send([{
  name: 'Default product',
  description: 'product description',
  price: 100
}]));
```

Como vimos no capítulo sobre os *middlewares* do *express*, os objetos de requisição (***req***) e resposta (***res***) são injetados automaticamente pelo *express* nas rotas. No caso acima usamos o método *send* do objeto de resposta para enviar uma lista com um produto como resposta da requisição, o que deve ser suficiente para que nosso teste passe.
Com as alterações o *app.js* deve estar assim:

```
import express from 'express';
import bodyParser from 'body-parser';

const app = express();
app.use(bodyParser.json());

app.get('/', (req, res) => res.send('Hello World!'));
app.get('/products', (req, res) => res.send([{
  name: 'Default product',
  description: 'product description',
  price: 100
}]));

export default app;
```

Agora que já temos a implementação, podemos executar nosso teste novamente:

```
$ npm run test:integration
```

A saída deve ser de sucesso, como essa:

```
Routes: Products
GET /products
✓ should return a list of products


1 passing (164ms)
```

Nosso teste está passando, e estamos no estado ***GREEN*** do *TDD*, ou seja, temos o teste e a implementação suficiente para ele passar. O próximo passo será o ***REFACTOR*** onde iremos configurar as rotas.

O código dessa etapa está disponível neste link[37].

[37] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step3

Estrutura de diretórios e arquivos

Um dos primeiros desafios quando começamos uma aplicação em *Node.js* é estruturar o projeto. Uma grande conveniência do *Node*, por ser *javascript*, é a liberdade para estrutura, *design* de código, *patterns* e etc. Porém, isso também pode gerar confusão para os novos desenvolvedores.

A maioria dos projetos no *github*[38], por exemplo, possuem estruturas que diferem entre si, essa variação acontece pois cada desenvolvedor cria a estrutura da forma que se enquadrar melhor a sua necessidade.

Mesmo assim podemos aproveitar os padrões comuns entre esses projetos para estruturar nossa aplicação de maneira que atenda as nossas necessidades e também fique extensível, legível e facilmente integrável com ferramentas externas, como *Travis*, *CodeClimate* e etc.

O diretório raiz

O diretório raiz do projeto é o ponto de entrada e fornece a primeira impressão. No exemplo a seguir, temos uma estrutura comum em aplicações usando o *framework Express.js*:

```
1  ├── app.js
2  ├── controllers
3  ├── middlewares
4  ├── models
5  ├── package.json
6  └── tests
```

Essa estrutura é legível e organizada, mas com o crescimento da aplicação pode misturar diretórios de código com diretórios de teste, *build* e etc. Um padrão comum em diversas linguagens é armazenar o código da aplicação em um diretório *source* normalmente chamado *src*.

[38] https://github.com

```
├── package.json
├── server.js
├── src
|   ├── controllers
|   ├── middlewares
|   └── models
|   └── app.js
└── tests
```

Dessa maneira o código da aplicação é isolado em um diretório deixando a raiz do projeto mais limpa e acabando com a mistura de diretórios de código com diretórios de testes e arquivos de configuração.

O que fica no diretório raiz?

No exemplo acima movemos o código da aplicação para o diretório *src* mas mantivemos o diretório *tests*, isso acontece porque testes são executados por linha de comando ou por outras ferramentas. Inclusive os *test runners* como *mocha* e *karma* esperam que o diretório *tests* esteja no diretório principal. Outros diretórios comumente localizados na raiz são *scripts* de suporte ou *build*, exemplos, documentação e arquivos estáticos. No exemplo abaixo vamos incrementar nossa aplicação com mais alguns diretórios:

```
├── env
|   ├── dev.env
|   └── prod.env
├── package.json
├── public
|   ├── assets
|   ├── css
|   ├── images
|   └── js
├── scripts
|   └── deploy.sh
├── server.js
├── src
|   ├── app.js
|   ├── controllers
|   ├── middlewares
|   ├── models
|   └── routes
└── tests
```

O diretório *public* é responsável por guardar tudo aquilo que vai ser entregue para o usuário. Mantê-lo na raiz facilita a criação de rotas de acesso e a movimentação dos assets, caso necessário. Os diretórios *scripts* e *env* são relacionados a execução da aplicação e serão chamados por alguma linha de comando ou ferramenta externa, colocá-los em um diretório acessível promove a usabilidade.

Separação da execução e aplicação

No segundo passo, quando movemos o código para o diretório *src*, criamos um arquivo chamado **app.js** e mantemos o **server.js** no diretório raiz, dessa maneira deixamos o *server.js* com a responsabilidade de chamar o *app.js* e inicializar a aplicação. Assim isolamos a aplicação da execução e deixamos que ela seja executada por quem chamar, nesse caso o *server.js*, mas também poderia ser um módulo, como o *supertest*, que vai fazer uma abstração *HTTP* para executar os testes e acessar as rotas.

Dentro do diretório *source*

Agora que já entendemos o que fica fora do diretório *src* vamos ver como organizá-lo baseado nas nossas necessidades.

```
1 ├── src
2 |   ├── app.js
3 |   ├── controllers
4 |   ├── middlewares
5 |   ├── models
6 |   └── routes
```

Essa estrutura é bastante utilizada, ela é clara e separa as responsabilidades de cada componente, além de permitir o carregamento dinâmico.

Responsabilidades diferentes dentro de um mesmo source

Ás vezes, quando começamos uma aplicação, já sabemos o que será desacoplado e queremos dirigir nosso *design* para que no futuro seja possível separar e tornar parte do código um novo módulo. Outra necessidade comum é ter *APIs* específicas para diferentes tipos de clientes, como no exemplo a seguir:

```
└── src
    ├── app.js
    ├── mobile
    │   ├── controllers
    │   ├── index.js
    │   ├── middlewares
    │   ├── models
    │   └── routes
    └── web
        ├── controllers
        ├── index.js
        ├── middlewares
        ├── models
        └── routes
```

Esse cenário funciona bem mas pode dificultar o reúso de código entre os componentes. Então, antes de implementar, tenha certeza que seu caso de uso permite a separação dos clientes sem que um dependa do outro.

Server e *client* no mesmo repositório

Muitas vezes temos o *backend* e o *front-end* separados mas versionados juntos, no mesmo repositório, seja ele *git*, *mercurial*, ou qualquer outro controlador de versão. A estrutura mais comum que pude observar na comunidade para esse tipo de situação é separar o *server* e o *client* como no exemplo abaixo:

```
├── client
│   ├── controllers
│   ├── models
│   └── views
├── client.js
├── config
├── package.json
├── server
│   ├── controllers
│   ├── models
│   └── routes
├── server.js
└── tests
```

Essa estrutura é totalmente adaptável ás necessidades. No exemplo acima, os testes de ambas as aplicações estão no diretório *tests* no diretório raiz. Assim, se o projeto for adicionado em uma

integração contínua ele vai executar a bateria de testes de ambas as aplicações. O **server.js** e o **client.js** são responsáveis por iniciar as respectivas aplicações. Podemos ter um *npm start* no *package.json* que inicie os dois arquivos juntos.

Separação por funcionalidade

Um padrão bem frequente é o que promove a separação por funcionalidade. Nele abstraímos os diretórios baseado nas funcionalidades e não nas responsabilidades, como no exemplo abaixo:

```
└── src
    ├── app.js
    ├── orders
    |   ├── orders.controller.js
    |   └── orders.routes.js
    └── products
        ├── products.controller.js
        ├── products.model.js
        └── products.routes.js
```

Essa estrutura possui uma boa legibilidade e escalabilidade, por outro lado, pode crescer muito tornando o reúso de componentes limitado e dificultando o carregamento dinâmico de arquivos.

Conversão de nomes

Quando separamos os diretórios por suas responsabilidades pode não ser necessário deixar explícito a responsabilidade no nome do arquivo.

Veja o exemplo abaixo:

```
└── src
    ├── controllers
    |   └── products.js
    └── routes
        └── products.js
```

Como o nosso diretório é responsável por informar qual a responsabilidade dos arquivos que estão dentro dele, podemos nomear os arquivos sem adicionar o sufixo _ + nome do diretório (por exemplo: "_controller"). Além disso, o *javascript* permite nomear um módulo quando o importamos, permitindo que mesmo arquivos com o mesmo nome sejam facilmente distinguidos por quem está lendo o código, veja o exemplo:

```
1 Import ProductsController from './src/controllers/products';
2 Import ProductsRoute from './src/routes/products';
```

Dessa maneira não adicionamos nenhuma informação desnecessária ao nomes dos arquivos e ainda mantemos a legibilidade do código.

No decorrer do livro utilizaremos o exemplo seguindo o padrão *MVC* com a diretório *source* e os demais diretórios dentro, como *controllers*, *models* e etc.

Rotas com o *express router*

O *express* possui um *middleware* nativo para lidar com rotas, o ***Router***. O *Router* é responsável por administrar as rotas da aplicação e pode ser passado como parâmetro para o *app.use()*. Utilizando o *Router* é possível desacoplar as rotas e remover a necessidade de usar o *app* (instância do *express*) em outros lugares da aplicação.

Separando as rotas

Vamos alterar nossa aplicação para separar as rotas do *app*. Para isso devemos criar um diretório chamado ***routes*** dentro de *src*. Os diretórios devem ficar assim:

```
├── package.json
├── server.js
├── src
|   ├── app.js
|   └── routes
```

Dentro de *routes* criaremos um arquivo chamado *index.js* que será responsável por carregar todas as rotas da aplicação:

```
import express from 'express';

const router = express.Router();

export default router;
```

No código acima importamos o *express*, acessamos o *Router* dentro dele e depois o exportamos. Agora que temos um arquivo para administrar as rotas podemos mover a lógica de administração das rotas que estão no *app.js* para o nosso *index.js*. Primeiro movemos a rota padrão. O arquivo de rotas deverá ficar assim:

```
import express from 'express';

const router = express.Router();

router.get('/', (req, res) => res.send('Hello World!'));

export default router;
```

Rotas por recurso

No código anterior não movemos a rota *products* porque ela não ficará no *index.js*. Cada recurso da *api* terá seu próprio arquivo de rotas e o *index.js* será responsável por carregar todos eles.

Agora, vamos criar um arquivo para as rotas do recurso *products* da nossa *api*.

Para isso será necessário criar um arquivo chamado ***products.js*** dentro do diretório *routes*, ele terá o seguinte código:

```
import express from 'express';

const router = express.Router();

export default router;
```

Agora podemos mover a rota *products* do *app.js* para o *products.js*. Ele deve ficar assim:

```
import express from 'express';

const router = express.router();

router.get('/', (req, res) => res.send([{
  name: 'default product',
  description: 'product description',
  price: 100
}]));

export default router;
```

Note que agora o padrão da rota não é mais ***/products*** e somente / , isso é uma boa prática para separar recursos da *api*. Como nosso arquivo é *products.js* as rotas dentro dele serão referentes ao recurso *products* da *api*, assim internamente não precisamos repetir esse prefixo, deixaremos para o index carregar essa rota e dar o prefixo pra ela. Vamos alterar o *index.js* para carregar a nossa nova rota, ele deve ficar assim:

```
import express from 'express';
import productsRoute from './products';

const router = express.Router();

router.use('/products', productsRoute);
router.get('/', (req, res) => res.send('Hello World!'));

export default router;
```

Primeiro importamos a rota que foi criada anteriormente e damos o nome de ***productsRoute***. Depois, para carregar a rota, chamamos a função *use* do *router* passando o prefixo da rota que será */products* e o *productsRoute* que importamos.

Com as rotas configuradas, o último passo é alterar o *app.js* para carregar nosso arquivo de rotas, ele deve ficar assim:

```
import express from 'express';
import bodyParser from 'body-parser';
import routes from './routes';

const app = express();
app.use(bodyParser.json());
app.use('/', routes);

export default app;
```

As rotas que estavam no *app.js* foram movidas para seus respectivos arquivos e agora importamos apenas o *routes*. Como foi criado um *index.js* dentro de *routes* não é necessário especificar o nome do arquivo, apenas importar o diretório */routes* e automaticamente o módulo do *Node.js* procurará primeiro por um arquivo *index.js* e o importará. Em seguida o *routes* é passado como parâmetro para a função *use* junto com o /, significa que toda requisição vai ser administrada pelo *routes*.

Router paths

Nos passos anteriores foram criadas algumas rotas que simbolizam caminhos na aplicação combinando um padrão e um método *HTTP*, por exemplo, uma requisição do tipo *get* na rota / irá retornar *Hello World*, já em */products* irá retornar um produto *fake*. Essa é a maneira de definir *endpoints* em *APIs* com o *express router*.

O caminho passado por parâmetro para o método *HTTP* é chamado de *path*, por exemplo *router.get("/products")*. *Paths* podem ser *strings*, *patterns* ou expressões regulares. Caso precise testar rotas complexas o *express* possui um testador de rotas *online* onde é possível adicionar o caminho e verificar como ele será interpretado pelo *express router*.

Executando os testes

Nesse momento nossos testes devem estar passando novamente, o que irá nos garantir que nossa refatoração foi concluída com sucesso.

O código dessa etapa está disponivel aqui[39]

[39] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step4

Controllers

Os *controllers* serão responsáveis por receber as requisições das rotas, interagir com o *Model* quando necessário e retornar a resposta para o usuário. No nosso código atual, as rotas estão com muita responsabilidade e difíceis de testar isoladamente, pois dependemos do *express*. Para corrigir esse comportamento precisamos adicionar os *controllers*. Vamos criar os *controllers* guiados por testes de unidade, assim será possível validar o comportamento de forma separada do nosso sistema em si.

Configurando os testes de unidade

Como vimos no capítulo de testes de unidade[40], testes de unidade servem para testar pequenas partes do *software* isoladamente.

Para começar, crie um diretório chamado ***unit*** dentro do diretório *test*, na raiz do projeto. Assim como fizemos nos testes de integração, criaremos os arquivos de configuração para os testes. Vamos criar um arquivo chamado ***helpers.js*** dentro de *unit*, com o seguinte código:

```
1  import chai from 'chai';
2
3  global.expect = chai.expect;
```

Em seguida, vamos criar o arquivo ***mocha.opts*** para as configurações do *Mocha*. Ele deve possuir o seguinte código:

```
1  --require test/unit/helpers.js
2  --reporter spec
3  --compilers js:babel-core/register
4  --slow 5000
```

A última etapa de configuração dos testes de unidade será a criação de um comando para executar os testes. Vamos adicionar o seguinte *script* no *package.json*:

```
1  "test:unit": "NODE_ENV=test mocha --opts test/unit/mocha.opts test/unit/**/*_spe\
2  c.js"
```

Para testar se o comando esta funcionando basta executar:

[40] https://github.com/waldemarnt/building-testable-apis-with-nodejs/blob/master/book/build.md#testes-de-unidade-unit-tests

```
$ npm run test:unit
```

A saída do terminal deve informar que não conseguiu encontrar os arquivos de teste:

```
Warning: Could not find any test files matching pattern: test/unit/**/*_spec.js
No test files found
```

Vamos criar nosso primeiro teste de unidade para o nosso futuro *controller* de produtos. A separação de diretórios será semelhante a da aplicação com *controllers*, *models* e etc.

Criaremos um diretório chamado *controllers* dentro de *unit* e dentro dele um arquivo com o cenário de teste que vamos chamar de *products_spec.js*. Em seguida vamos executar os testes unitários novamente, a saída deve ser a seguinte:

```
0 passing (2ms)
```

Ok, nenhum teste está passando pois ainda não criamos nenhum.

Testando o *controller* unitariamente

Vamos começar a escrever o teste. O primeiro passo será adicionar a descrição desse cenário de testes, como no código a seguir:

```
describe('Controllers: Products', () => {

});
```

Esse cenário irá englobar todos os testes do *controller* de *products*. Vamos criar cenários para cada um dos métodos, o próximo cenário terá o seguinte código:

```
describe('Controllers: Products', () => {

    describe('get() products', () => {

    });

});
```

Precisamos agora criar nosso primeiro caso de teste para o método *get*. Começaremos nosso caso de teste descrevendo o seu comportamento:

```
describe('Controllers: Products', () => {

  describe('get() products', () => {
    it('should return a list of products', () => {

    });
  });

});
```

Segundo a descrição do nosso teste, o método *get* deve retornar uma lista de produtos. Esse é o comportamento que iremos garantir que está sendo contemplado. Começaremos iniciando um novo *controller* como no código a seguir:

```
import ProductsController from '../../../src/controllers/products';

describe('Controllers: Products', () => {

  describe('get() products', () => {
    it('should return a list of products', () => {

      const productsController = new ProductsController();

    });
  });
});
```

Importamos o *ProductsController* do diretório onde ele deve ser criado e dentro do caso de teste inicializamos uma nova instância. Nesse momento se executarmos nossos testes de unidade devemos receber o seguinte erro:

```
Error: Cannot find module '../../../src/controllers/products'
    at Function.Module._resolveFilename (module.js:455:15)
    at Function.Module._load (module.js:403:25)
…
```

A mensagem de erro explica que o módulo *products* não foi encontrado, como esperado. Vamos criar o nosso *controller* para que o teste passe. Vamos adicionar um diretório chamado ***controllers*** em *src* e dentro dele vamos criar o arquivo ***products.js***, que será o *controller* para o recurso de *products* da *API*:

```
class ProductsController {

}

export default ProductsController;
```

Com o *controller* criado no diretório correto o nosso teste deve estar passando, vamos tentar novamente os testes unitários:

```
$ npm run test:unit
```

A saída do terminal deve ser a seguinte:

```
  Controllers: Products
    get() products
      ✓ should return a list of products


  1 passing (176ms)
```

Até o momento ainda não validamos o nosso comportamento esperado, apenas foi validado que o nosso *controller* existe. Agora precisamos garantir que o comportamento esperado no teste está sendo coberto, para isso precisamos testar se o método *get* chama a função de resposta do *express*. Antes de começar esse passo precisamos instalar o *Sinon*, uma biblioteca que irá nos ajudar a trabalhar com *spies*, *stubs* e *mocks*, os quais serão necessários para garantir o isolamento dos testes unitários.

Mocks, Stubs e *Spies* com *Sinon.js*

Para instalar o *Sinon* basta executar o seguinte comando:

```
$ npm install --save-dev sinon
```

Após a instalação ele já estará disponível para ser utilizado em nossos testes. Voltando ao teste, vamos importar o *Sinon* e também usar um *spy* para verificar se o método *get* do *controller* está realizando o comportamento esperado. O código do teste deve ficar assim:

```
1  import ProductsController from '../../../src/controllers/products';
2  import sinon from 'sinon';
3
4  describe('Controllers: Products', () => {
5    const defaultProduct = [{
6      name: 'Default product',
7      description: 'product description',
8      price: 100
9    }];
10
11   describe('get() products', () => {
12     it('should return a list of products', () => {
13       const request = {};
14       const response = {
15         send: sinon.spy()
16       };
17
18       const productsController = new ProductsController();
19       productsController.get(request, response);
20
21       expect(response.send.called).to.be.true;
22       expect(response.send.calledWith(defaultProduct)).to.be.true;
23     });
24   });
25 });
```

Muita coisa aconteceu nesse bloco de código, mas não se preocupe, vamos passar por cada uma das alterações.

A primeira adição foi o *import* do *Sinon*, módulo que instalamos anteriormente.

Logo após a descrição do nosso cenário de teste principal adicionamos uma *constant* chamada ***defaultProduct*** que armazena um *array* com um objeto referente a um produto com informações estáticas. Ele será útil para reaproveitarmos código nos casos de teste.

Dentro do caso de teste foram adicionadas duas *constants*: *request*, que é um objeto *fake* da requisição enviada pela rota do *express*, que vamos chamar de *req* na aplicação, e *response*, que é um objeto *fake* da resposta enviada pela rota do *express*, a qual vamos chamar de *res* na aplicação.

Note que a propriedade *send* do objeto *response* recebe um *spy* do *Sinon*, como vimos anteriormente, no capítulo de *test doubles*, os *spies* permitem gravar informações como quantas vezes uma função foi chamada, quais parâmetros ela recebeu e etc. O que será perfeito em nosso caso de uso pois precisamos validar que a função *send* do objeto response está sendo chamada com os devidos parâmetros.

Até aqui já temos a configuração necessária para reproduzir o comportamento que esperamos. O próximo passo é chamar o método *get* do *controller* passando os objetos *request* e *response* que criamos. E o último passo é verificar se o método *get* está chamando a função *send* com o *defaultProduct* como parâmetro. Para isso foram feitas duas asserções, a primeira verifica se a função *send* foi chamada, e a segunda verifica se ela foi chamada com o *defaultProduct* como parâmetro.

Nosso teste está pronto, se executarmos os testes unitários devemos receber o seguinte erro:

```
  Controllers: Products
    get() products
      1) should return a list of products


  0 passing (156ms)
  1 failing

  1) Controllers: Products get() products should return a list of products:
     TypeError: productsController.get is not a function
      at Context.it (test/unit/controllers/products_spec.js:19:26)
```

O erro explica que *productsController.get* não é uma função, então vamos adicionar essa função ao *controller*. A função *get* deverá possuir a lógica que agora está na rota de produtos. Vamos adicionar o método *get* no *ProductsController*, o código deve ficar assim:

```
class ProductsController {

  get(req, res) {
    return res.send([{
      name: 'Default product',
      description: 'product description',
      price: 100
    }])
  }
}

export default ProductsController;
```

O método *get* deve receber os objetos de requisição e resposta e enviar um *array* com um produto estático como resposta.

Vamos executar os testes novamente, a saída do terminal deve ser a seguinte:

```
  Controllers: Products
    get() products
      ✓ should return a list of products


  1 passing (189ms)
```

Integrando *controllers* e rotas

Nosso *controller* está feito e estamos obtendo o comportamento esperado, mas até então não integramos com a aplicação. Para realizar essa integração basta alterar a rota de produtos para usar o *controller*. Edite o arquivo *products.js* em *src/routes*, removendo o bloco de código que foi movido para o *controller*, e adicione a chamada para o método *get*. A rota de produtos deve ficar assim:

```
import express from 'express';
import ProductsController from '../controllers/products';

const router = express.Router();
const productsController = new ProductsController();
router.get('/', (req, res) => productsController.get(req, res));

export default router;
```

Vamos executar os testes de integração para garantir que o *controller* foi integrado corretamente com o resto da nossa aplicação.

```
$ npm run test:integration
```

A saída do terminal deve ser a seguinte:

```
  Routes: Products
    GET /products
      ✓ should return a list of products


  1 passing (251ms)
```

Os código desta etapa esta disponível aqui[41]

[41] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step5

Configurando o *MongoDB* como banco de dados

/todo

Introdução ao *MongoDB*

/todo

Configurando o banco de dados com *Mongoose*

Para integrar nossa aplicação com o *MongoDB* vamos utilizar o *Mongoose*[42] que é um *ODM (Object Document Mapper)*. O *Mongoose* irá abstrair o acesso ao banco de dados e ainda irá se responsabilizar por transformar os dados do banco em *Models*, facilitando a estruturação de nossa aplicação com o padrão *MVC*.

Para instalar o *Mongoose* basta executar o seguinte comando *npm*:

```
1 $ npm install mongoose --save
```

Após a instalação o *Mongoose* estará disponível para ser utilizado. O próximo passo será configurar a aplicação para conectar com o banco de dados. Para isso vamos criar um diretório chamado *config* dentro de *src* e dentro dele um arquivo chamado ***database.js*** que será responsável por toda configuração do banco de dados.

A estrutura de diretórios deve estar assim:

[42] http://mongoosejs.com/

```
├── src
│   ├── app.js
│   ├── config
│   │   └── database.js
│   ├── controllers
│   │   └── products.js
│   └── routes
│       ├── index.js
│       └── products.js
```

A primeira coisa que deve ser feita no *database.js* é importar o módulo do *Mongoose*, como no código abaixo:

```
import mongoose from 'mongoose';
```

O próximo passo será informar qual biblioteca de *Promises* será utilizada. Essa é uma necessidade da versão 4 do *Mongoose*, como estamos utilizando o *Node.js* na versão 6 que já conta com *Promises* por padrão, será necessário fazer somente o seguinte:

```
mongoose.Promise = Promise;
```

Aqui estamos dizendo para o *Mongoose* utilizar a *Promise* oficial do *Node.js*.

Seguindo a configuração do banco de dados é necessário informar a *url* onde está o *MongoDB*. No meu caso está no meu computador então a *url* será *localhost* seguido do nome que daremos ao banco de dados:

```
const mongodbUrl = process.env.MONGODB_URL || 'mongodb://localhost/test';
```

Note que primeiro verificamos se não existe uma variável de ambiente, caso não exista é usado o valor padrão que irá se referir ao *localhost* e ao banco de dados *test*. Dessa maneira, poderemos utilizar o *MongoDB* tanto para testes quanto para rodar o banco da aplicação, sem precisar alterar o código.

No próximo passo vamos criar uma função para conectar com o banco de dados:

```
const connect = () => mongoose.connect(mongodbUrl);
```

Acima, criamos uma função que retorna uma conexão com o *MongoDB*, esse retorno é uma *Promise*, ou seja, somente quando a conexão for estabelecida a *Promise* será resolvida. Isso é importante pois precisamos garantir que nossa aplicação só vai estar disponível depois que o banco de dados estiver conectado e acessível.

O último passo é exportar o módulo de configuração do banco de dados:

```
export default {
  connect
}
```

O código do *database.js* deve estar similar ao que segue:

```
import mongoose from 'mongoose';

mongoose.Promise = Promise;

const mongodbUrl = process.env.MONGODB_URL || 'mongodb://localhost/test';

const connect = () => mongoose.connect(mongodbUrl);

export default {
  connect
}
```

Pronto, o banco de dados está configurado. Nosso próximo passo será integrar o banco de dados com a aplicação, para que ela inicialize o banco sempre que for iniciada.

Integrando o *Mongoose* com a aplicação

O módulo responsável por inicializar a aplicação é o *app.js*, então, ele que vai garantir que o banco estará disponível para que a aplicação possa consumi-lo. Vamos alterar o ***app.js*** para que ele integre com o banco de dados, atualmente ele está assim:

```
import express from 'express';
import bodyParser from 'body-parser';
import routes from './routes';

const app = express();
app.use(bodyParser.json());
app.use('/', routes);

export default app;
```

O primeiro passo é importar o módulo responsável pelo banco de dados, o *database.js*, que fica dentro do diretório *config*. Os *imports* devem ficar assim:

```
 import express from 'express';
 import bodyParser from 'body-parser';
 import routes from './routes';
+import database from './config/database';
```

A seguir vamos alterar um pouco o código anterior que utiliza o *express* e as rotas movendo o seguinte trecho:

```
- app.use(bodyParser.json());
- app.use('/', routes);

- export default app;
```

Os trechos em vermelho serão movidos para dentro de uma nova função, como no código abaixo:

```
+ const configureExpress = () => {
+  app.use(bodyParser.json());
+  app.use('/', routes);
+
+  return app;
+};
```

Acima criamos uma função nomeada ***configureExpress*** que terá a tarefa de configurar o *express* e retornar uma nova instância de aplicação configurada.

A última etapa da nossa alteração é inicializar o banco antes da aplicação. Como o *moongose* retorna uma *Promise*, vamos esperar ela ser resolvida para então chamar a função que criamos anteriormente e que configura o *express*:

```
+ export default () => database.connect().then(configureExpress);
```

No bloco acima exportamos uma função que retorna uma *Promise*. A primeira chamada é a função *connect* do *database*, que criamos na etapa anterior, assim que essa *Promise* for resolvida, significa que o banco de dados estará disponível, então é chamada a função *configureExpress* que irá configurar o *express* e retornar uma nova instância da aplicação. Esse *pattern* é conhecido como *chained promises*.

Note que a função *configureExpress* não precisaria existir, poderíamos ter uma função diretamente dentro do *then* do *connect* e nela configurar o *express*, porém criar uma função que descreva o que está sendo feito torna o código mais claro e desacoplado. Pode se ler mais sobre o assunto nesta *issue* do *airbnb*[43].

O *app.js* depois de alterado deve estar assim:

[43] https://github.com/airbnb/javascript/issues/216

```
import express from 'express';
import bodyParser from 'body-parser';
import routes from './routes';
import database from './config/database'

const app = express();

const configureExpress = () => {
  app.use(bodyParser.json());
  app.use('/', routes);

  return app;
};

export default () => database.connect().then(configureExpress);
```

Como alteramos o *app* para retornar uma função, que por sua vez retorna uma *Promise*, será necessário alterar o *server.js* para fazer a inicialização de maneira correta.

Alterando a inicilização

O *server.js* é o arquivo responsável por inicializar a aplicação, chamando o *app*. Como alteramos algumas coisas na etapa anterior precisamos atualizá-lo. Vamos começar alterando o nome do módulo na importação:

```
- import app from './src/app';
+ import setupApp from './src/app';
```

O módulo foi alterado de *app* para *setupApp*, por quê? Porque agora ele é uma função e esse nome reflete mais a sua responsabilidade.

O próximo passo é alterar a maneira como o *app* é chamado:

```
-app.listen(port, () => {
-  console.log(`app running on port ${port}`);
-});
+setupApp()
+ .then(app => app.listen(port, () => console.log(`app running on port ${port}`)\
))
+ .catch(error => {
+   console.error(error);
+   process.exit(1);
+ });
```

Como o código anterior devolvia uma instância da aplicação diretamente, era apenas necessário chamar o método *listen* do *express* para inicializar a aplicação. Agora temos uma função que retorna uma *promise* devemos chamá-la e ela vai inicializar o *app*, inicializando o banco, configurando o *express* e retornando uma nova instância da aplicação, só então será possível inicializar a aplicação chamando o *listen*.

Até esse momento espero que vocês já tenham lido a especificação de *Promises* mais de 10 vezes e já sejam mestres na implementação. Quando um problema ocorre a *Promise* é rejeitada. Esse erro pode ser tratado usando um *catch* como no código acima. Acima, recebemos o erro e o mostramos no *console.log*, em seguida encerramos o processo do *Node.js* com o código 1. Dessa maneira o processo é finalizado informando que houve um erro em sua inicialização. Informar o código de saída, é uma boa prática finalizar o processo com código de erro e conhecido como *graceful shutdowns* e faz parte da lista do *12 factor app*[44] de boas práticas para desenvolvimento de *software* moderno.

As alterações necessárias para integrar com o banco de dados estão finalizadas, vamos executar os testes de integração para garantir:

```
$ npm run test:integration
```

A saida será:

```
 Routes: Products
    GET /products
      1) should return a list of products


  0 passing (152ms)
  1 failing

  1) Routes: Products GET /products should return a list of products:
     TypeError: Cannot read property 'get' of undefined
      at Context.done (test/integration/routes/products_spec.js:21:7)
```

[44] https://12factor.net/

O teste quebrou! Calma, isso era esperado. Assim como o *server.js* o teste de integração inicia a aplicação usando o módulo *app*, então ele também deve ser alterado para lidar com a *Promise*.

Vamos começar alterando o ***helpers.js*** dos testes de integração, como no código abaixo:

```
-import app from '../../src/app.js';
+import setupApp from '../../src/app.js';

-global.app = app;
-global.request = supertest(app);
+global.setupApp = setupApp;
+global.supertest = supertest;
```

Assim como no *server.js*, alteramos o nome do módulo de *app* para *setupApp* e o exportamos globalmente. Também removemos o *request* do conceito global que era uma instância do *supertest* com o *app* configurado, deixaremos para fazer isso no próximo passo.

Agora é necessário alterar o ***products_spec.js*** para inicializar a aplicação antes de começar a executar os casos de teste usando o *callback before* do *Mocha*:

```
describe('Routes: Products', () => {
+  let request;
+
+  before(()=> {
+    return setupApp()
+      .then(app => {
+        request = supertest(app)
+      })
+  });
+
```

No bloco acima, criamos um *let* para o *request* do *supertest* e no *before* a aplicação é inicializada. Assim que o *setupApp* retornar uma instância da aplicação é possível inicializar o *supertest* e atribuir a *let request* que definimos anteriormente.

let no lugar de *const*

let e *const* são novos tipos de declaração de variáveis do *Ecmascript* 6 ambas possuem comportamento similar mas com uma grande diferença, *constants* não podem ter seu valor alterado. Por isso foi utilizado *let* no código acima, pois o valor precisa ser reescrito após o *before*. Leia mais sobre *let*, *const* e *block bindings* aqui[45].

Executando os testes novamente, a saída deve ser a seguinte:

[45] http://walde.co/2016/05/13/javascript-es6-let-e-const-e-block-bindings/

```
  Routes: Products
    GET /products
      ✓ should return a list of products


  1 passing (336ms)
```

Caso ocorra um erro como: *"MongoError: failed to connect to server [localhost:27017] on first connect"*:

```
  Routes: Products
    1) "before all" hook


  0 passing (168ms)
  1 failing

  1) Routes: Products "before all" hook:
     MongoError: failed to connect to server [localhost:27017] on first connect
      at Pool.<anonymous> (node_modules/mongodb-core/lib/topologies/server.js:32\
6:35)
      at Connection.<anonymous> (node_modules/mongodb-core/lib/connection/pool.j\
s:270:12)
      at Socket.<anonymous> (node_modules/mongodb-core/lib/connection/connection\
.js:175:49)
      at emitErrorNT (net.js:1272:8)
      at _combinedTickCallback (internal/process/next_tick.js:74:11)
      at process._tickCallback (internal/process/next_tick.js:98:9)
```

A mensagem de erro explica que o *MongoDB* não está executando em *localhost* na porta 7000, verifique e tente novamente.

O código desta etapa esta disponível aqui[46].

[46] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step6

O padrão *MVC*

MVC é um *design pattern* de arquitetura que foca na separação de responsabilidades. O *MVC* separa os dados de negócio (*Models*) da interface do usuário (*Views*) e usa um componente para ligá-los (*Controllers*), normalmente os *controllers* são responsáveis por receber a entrada do usuário e coordenar *Models* e *Views*.

Voltando ao tempo do *Smalltalk*

Para entender o padrão *MVC* é necessário voltar alguns anos no tempo, aos anos 70 para ser mais preciso, fase da emergência das interfaces gráficas de usuário. Na década de 70, *Trygve Reenskaug*[47] trabalhava em uma linguagem de programação chamada *Smalltalk*[48], com a qual ele desenvolveu e aplicou o padrão *MVC* com objetivo de separar a interface do usuário da lógica da aplicação, conceito conhecido como *Separated Presentation*[49]. Essa arquitetura consistia em:

- Um elemento de domínio denominado *Model* que não deveria ter conhecimento da interface e interação do usuário (*Views* e *Controllers*).
- A camada de apresentação e interação seria feita pelas *views* e *controllers*, com um *controller* e uma *view* para cada elemento que seria mostrado na tela.
- A responsabilidade do *controller* era receber entradas das *views*, como por exemplo: teclas pressionadas, formularios ou eventos de *click*.
- Para atualizar a *view* era utilizado um padrão conhecido como *Observer*. Cada vez que o *model* mudava, essa mudança era refletida na *view*.

É impressionante ver que um padrão utilizando até hoje como o *Observer*[50] (também implementado como *pub/sub* hoje em dia) era parte crucial do *MVC*. No *Smalltak MVC* ambos *views* e *controllers* observavam o *model*, para que qualquer mudança conseguisse ser refletida na *view*. Para quem deseja se aprofundar mais nas origens do padrão *MVC* indico o artigo *GUI Architectures*[51], do *Martin Fowler*, que conta a história do *MVC* ao longo dos anos.

[47] https://en.wikipedia.org/wiki/Trygve_Reenskaug
[48] https://en.wikipedia.org/wiki/Smalltalk
[49] https://martinfowler.com/eaaDev/uiArchs.html
[50] https://addyosmani.com/resources/essentialjsdesignpatterns/book/#observerpatternjavascript
[51] https://martinfowler.com/eaaDev/uiArchs.html

MVC no *javascript*

Nos anos de hoje o *MVC* é base para a maioria dos *frameworks* e arquiteturas de projetos em diversas linguagens. *Frameworks* como *SpringMVC* do *Java*, *Symfony* do PHP e *Ruby on Rails* do *Ruby* utilizam esse padrão para o desenvolvimento.

No *javascript* o *MVC* começou no contexto dos *browsers*, quando as aplicações começaram a ter mais responsabilidades no *front-end* com as *single page apps* o *javascript* passou pela mesma necessidade do *Smalltalk* nos anos 70: separar a interface da lógica de negócio. Uma gama de *frameworks* para o *front-end* apareceram aplicando *MVC* com diversas variações, como por exemplo o *Backbone* e o *Ember.js*. A chegada do *javascript* no *server side* com o *Node.js* trouxe a mesma necessidade, *frameworks* como *Express.js* e *Sails.js* utilizam *MVC* como base de arquitetura para criação de aplicações.

Conforme mencionado, o *MVC* é composto por três componentes:

Models

Os *Models* são responsáveis por administrar os dados da aplicação. Os *models* variam muito de aplicações e *frameworks* mas normalmente eles são responsáveis por validar os dados e também persistir sincronizando com um *localStorage* ou banco de dados. *Models* podem ser observados por mais de uma *view*; *Views* podem precisar de partes diferentes dos dados de um *model*. Por exemplo, um *model* de usuário pode ser utilizado em uma *view* para mostrar nome e *email*, e em outra para mudar a senha.

Views

Views são uma forma visual de representar os *models* e apresentar a informaçao para o usuario. *Views* normalmente observam os *models* para refletirem suas mudanças na tela. A maioria dos livros de *design patterns* se referem à *views* como "burras", pois sua única responsabilidade é mostrar o estado do *model*.

Controllers

Controllers agem como mediadores entre *views* e *models*. Resumidamente, sua responsabilidade é atualizar o *model* quando a *view* muda e atualizar a *view* quando o *model* muda.

Para os desenvolvedores vindos do *javascript* no *front-end* há uma variação enorme[52] em como o *MVC* é implementado, muitas vezes omitindo o C, ou seja, não utilizando *controllers*, isso pois as

[52] https://www.quora.com/What-are-the-main-differences-between-MVC-MVP-and-MVVM-design-patterns-for-the-JavaScript-developer

necessidades no *front-end* são diferentes. Um exemplo de implementação de *controller* no *front-end* que sempre deu muita discussão é a do *Angular.js* versão 1.x, que é totalmente acoplado a *view* utilizando um padrão chamado de *2 way data binding*[53] que faz com que as alterações na tela sejam enviadas para o *controller* e as alterações no *controller* enviadas para a *view*. Além de ser responsável por escutar e emitir eventos, o que atribui muita responsabilidade para os *controllers*, assim quebrando o princípio de responsabilidade única[54].

Em 2015 com o surgimento do *React* e *frameworks* como *Flux*, o *MVC* no *front-end* perdeu força[55] pois a componentização, programação reativa e as arquiteturas unidirecionais começaram a fazer mais sentido no contexto dos *browsers*.

Já no *server side* com *Node.js* esse padrão ainda é muito utilizado e útil. No *Sails.js*[56] a implementação de um *controller* para administrar as requisições e conversar com o *model* é obrigatório. Já no *Express.js*[57] essa conversão não é obrigatória, mas é sugerida. O próprio gerador de código do *Express* já cria o diretório para *controllers* separadamente.

As vantagens de utilizar *MVC*

A separação de responsabilidades facilita a modularização das funcionalidades da aplicação e possibilita:

- Manutenibilidade: Quando uma modificação precisa ser feita é mais fácil de descobrir onde é e também onde vai causar impacto.
- Desacoplar *models* e *views* facilita os testes em ambos isoladamente. Testando a lógica de negócio em *models* e a usabilidade em *views*.
- Reutilização de código

MVC em *API*

Para *APIs*, a parte da *view* não é aplicada. No decorrer do livro será aplicado o padrão *MVC*, adaptado para o contexto da *API* que será desenvolvida, com *Controllers* e *Models*. Em *APIs* o *controller* recebe a requisição da rota, faz a chamada para o *model* realizar a lógica de negócio e retorna a resposta para o usuário.

[53] https://www.quora.com/What-is-two-way-data-binding-in-Angular
[54] http://www.oodesign.com/single-responsibility-principle.html
[55] https://medium.freecodecamp.com/is-mvc-dead-for-the-frontend-35b4d1fe39ec#.u1yvsd5lt
[56] http://sailsjs.com/
[57] https://expressjs.com/

Models

Como visto no capítulo sobre *MVC*, os *models* são responsáveis pelos dados, persistência e validação na aplicação. Aqui estamos utilizando o *Mongoose*, que já provê uma *API* para a utilização de *models*.

Criando o *model* com *Mongoose*

O primeiro passo será a criação de um diretório chamando ***models*** e um arquivo chamado ***products.js*** dentro de *src*, como no exemplo abaixo:

```
├── src
|   ├── models
|   |   └── product.js
```

No *products.js* devemos começar importando o módulo do *mongoose*:

```
import mongoose from 'mongoose';
```

Após isso será necessário descrever o *schema* do *model* de *products*. O *schema* é utilizado pelo *mongoose* para válidar e mapear os dados do *model*. Cada *schema* representa uma *collection* do *MongoDB*.

Adicione um *schema* como o seguinte:

```
const schema = new mongoose.Schema({
  name: String,
  description: String,
  price: Number
});
```

No bloco acima uma nova instância de *schema* é definida e atribuída a *constant schema*, o *model* está definido, agora basta exportá-lo para que ele possa ser utilizado na aplicação:

```
const Product = mongoose.model('Product', schema);

export default Product;
```

Chamando *mongoose.model* com um nome, no nosso caso *Product* definimos um *model* no módulo *global* do *mongoose*. O que significa que qualquer lugar que importar o módulo do *mongoose* a partir de agora na aplicação poderá acessar o *model* de *products* que foi definido pois o módulo do *mongoose* é um *Singleton*.

A versão final do *model Product* deve ficar similar a esta:

```
import mongoose from 'mongoose';

const schema = new mongoose.Schema({
  name: String,
  description: String,
  price: Number
});
const Product = mongoose.model('Product', schema);

export default Product;
```

Singleton Design Pattern

No *Node.js*, e no *Javascript* em geral, existem inúmeras maneiras de aplicar o *Singleton*, vamos revisar as formas mais utilizadas. Tradicionalmente o *Singleton* restringe a inicialização de uma nova classe a um único objeto ou referência. Segundo Addy Osmani, no livro *Javascript Design Patterns*[58]:

> With JavaScript, singletons serve as a namespace provider which isolate implementation code from the global namespace so-as to provide a single point of access for functions.

Traduzindo livremente:

Singletons em *javascript* servem como um provedor de *namespaces* isolando a implementação do código do *namespace* global possibilitando assim acesso a somente um ponto, que podem ser funções ou classes por exemplo.

No código a seguir definimos um *Model* no *Mongoose*:

[58] https://addyosmani.com/resources/essentialjsdesignpatterns/book/

```
import mongoose from 'mongoose';

const schema = new mongoose.Schema({
  name: String,
  description: String,
  price: Number
});
const Product = mongoose.model('Product', schema);

export default Product;
```

Note que importamos o módulo do *Mongoose* e não iniciamos uma nova instância com *new*, apenas acessamos o módulo diretamente. Em seguida, definimos um novo *schema* para o *Model* e então, utilizando a função *mongoose.model*, definimos um *model* chamado *Product* na instância do *mongoose* que importamos.

Agora se importarmos o módulo do *Mongoose* em qualquer outro lugar da aplicação e acessarmos os *models* teremos uma resposta como a seguinte:

```
//src/routes/products.js

import mongoose from 'mongoose';

console.log(mongoose.models);
```

O *console.log* mostrará:

```
{ Product:
   { [Function: model]
...
```

Essa é a implementação e a responsabilidade de um *Singleton*: prover acesso a mesma instância independente de quantas vezes ou da maneira que for chamada.

Vamos ver como é implementado o *Singleton* no código do *Mongoose*. No arquivo */lib/index.js* do módulo temos a seguinte função:

```
function Mongoose() {
  this.connections = [];
  this.plugins = [];
  this.models = {};
  this.modelSchemas = {};
  // default global options
  this.options = {
    pluralization: true
  };
  var conn = this.createConnection(); // default connection
  conn.models = this.models;
}
```

Para quem não é familiarizado com *es2015*, a função *Mongoose()* representará uma classe. No final do arquivo podemos ver como o módulo é exportado:

```
var mongoose = module.exports = exports = new Mongoose;
```

Essa atribuições: *var mongoose = module.exports = exports* não são o nosso foco. A parte importante dessa linha é o *new Mongoose* que garante que o módulo exportado será uma nova instância da classe *Mongoose*.

Você pode estar se perguntando se uma nova instância será criada sempre que importamos o módulo, a resposta é não. Módulos no *Node.js* são *cacheados* uma vez que carregados, o que significa que o que acontece no *module.exports* só acontecerá uma vez a cada inicialização da a aplicação ou quando o *cache* for limpo (o que só pode ser feito manualmente). Dessa maneira o código acima exporta uma referência a uma nova classe e quando a importamos temos acesso diretamente a seus atributos e funções internas.

Singletons são extremamente úteis para manter estado em memória possibilitando segurança entre o compartilhamento de uma mesma instância a todos que a importarem.

Veremos mais sobre módulos no capítulo sobre modularização.

Integrando *models* e *controllers*

Até agora nosso *controller* responde com dados *fakes* e nosso teste de integração ainda está no estado *GREEN*. Adicionamos o *model* e agora precisamos integrar ele com o *controller* e depois com a rota para que seja possível finalizar a integração e completar o passo de *REFACTOR* do nosso teste de integração.

Para começar vamos atualizar o teste de unidade do *controller* para refletir o comportamento que esperamos. Para isso devemos começar atualizando o arquivo *test/unit/controllers/products_spec.js* importando os módulos necessários para descrever o comportamento esperado no teste:

```
import ProductsController from '../../../src/controllers/products';
import sinon from 'sinon';
+import Product from '../../../src/models/product';
```

Aqui importamos o módulo referente ao model de *Product* que criamos anteriormente e será usado pelo *controller*.

Agora vamos mudar o caso de teste incluindo o comportamento que esperamos quando integrarmos com o *model*.

```
  describe('get() products', () => {
-   it('should return a list of products', () => {
+   it('should call send with a list of products', () => {
      const request = {};
      const response = {
        send: sinon.spy()
      };
+     Product.find = sinon.stub();
```

No código acima começamos atualizando a descrição, não iremos checar o retorno pois a saída da função *get* é uma chamada para a função *send* do *express*, então nossa descrição deve refletir isso, dizemos que: "Isso deve chamar a função *send* com uma lista de produtos".

Logo após atribuimos um *stub* para a função *find* do *model Product*. Desta maneira será possível adicionar qualquer comportamento para essa função simulando uma chamada de banco de dados por exemplo. O próximo passo será mudar o seguinte código para utilizar o *stub*:

```
-      const productsController = new ProductsController();
-      productsController.get(request, response);
+      Product.find.withArgs({}).resolves(defaultProduct);
```

No *withArgs({})* dizemos para o *stub* que quando ele for chamado com um objeto vazio ele deve resolver uma *Promise* retornando o *defaultProduct*. Esse comportamento será o mesmo que o *moongose* fará quando buscar os dados do banco de dados. Mas como queremos testar isoladamente vamos remover essa integração com o banco de dados utilizando esse *stub*.

Agora precisamos mudar o comportamento esperado:

```
-      expect(response.send.called).to.be.true;
-      expect(response.send.calledWith(defaultProduct)).to.be.true;
+      const productsController = new ProductsController(Product);
+      return productsController.get(request, response)
+        .then(() => {
+          sinon.assert.calledWith(response.send, defaultProduct);
+        });
+    });
```

No código acima, o primeiro passo foi iniciar uma nova instância de *ProductsController* passando por parâmetro o *model*. Dessa maneira esperamos que cada instância de *controller* possua um *model*. Na linha seguinte retornamos a função *get* do *productsController*. Isso por que ela será uma *Promise*, e precisamos retornar para que nosso *test runner*, o *Mocha*, a chame e a resolva. Quando a *Promise* é resolvida é checado se a função *send* do objeto *response*, que é um *spy*, foi chamada com o *defaultProduct*:

```
sinon.assert.calledWith(response.send, defaultProduct);
```

Isso valida que a função *get* foi chamada, chamou a função *find* do *model Product* passando um objeto vazio e ele retornou uma *Promise* contendo o *defaultProduct*. O código final deve estar similar a este:

```
import ProductsController from '../../../src/controllers/products';
import sinon from 'sinon';
import Product from '../../../src/models/product';

describe('Constrollers: Products', () => {
  const defaultProduct = [{
    name: 'Default product',
    description: 'product description',
    price: 100
  }];

  describe('get() products', () => {
    it('should call send with a list of products', () => {
      const request = {};
      const response = {
        send: sinon.spy()
      };
      Product.find = sinon.stub();

      Product.find.withArgs({}).resolves(defaultProduct);
```

```
21
22        const productsController = new ProductsController(Product);
23        return productsController.get(request, response)
24          .then(() => {
25            sinon.assert.calledWith(response.send, defaultProduct);
26          });
27      });
28
29    });
30  });
```

Se executarem os testes de unidade agora, eles estão falhando, então vamos a implementação!

Atualizando o *controller* para utilizar o *model*

Agora precisamos atualizar o *controller products* que fica em: *src/controllers/products.js*. Vamos começar adicionando um construtor para poder receber o *model Product*, como no código a seguir:

```
1  class ProductsController {
2  +  constructor(Product) {
3  +    this.Product = Product;
4  +  };
```

O construtor irá garantir que toda a vez que alguém tentar criar uma instância do *controller* ele deve passar o *model Product* por parâmetro. Mas ai vocês me perguntam, mas por que não importar ele diretamente no *productsController.js*? Pois assim não seria possível usar *stub* no *model* e tornaria o código acoplado. Veremos mais sobre como gerenciar dependencias nos capítulos seguintes.

Seguindo a atualização do *controller* agora devemos atualizar o método que estamos testando, o *get*. Como no código a seguir:

```
1     get(req, res) {
2  -    return res.send([{
3  -      name: 'Default product',
4  -      description: 'product description',
5  -      price: 100
6  -    }])
7  +    return this.Product.find({})
8  +      .then(products => res.send(products));
9      }
10   }
```

Aqui removemos o produto *fake* que era retornado, para aplicar a lógica real de integração com o banco. Note que *this.Product.find({})* segundo a documentação do *mongoose*[59] irá retornar uma lista de objetos, então o que está sendo feito quando a *Promise* resolver é passar essa lista para a função *send* do objeto *res* do *express* para que ele retorne para o usuário que fez a requisição.

Essa é a implementação necessária para que o teste passe, vamos rodá-lo:

```
$ npm run test:unit
```

A resposta deve ser:

```
  Controllers: Products
    get() products
      ✓ should call send with a list of products


  1 passing (217ms)
```

Testando casos de erro

Até agora apenas testamos o *happy path*[60] (termo usado para descrever o caminho feliz esperado em um teste), mas o que acontecerá se der algum erro na consulta ao banco? Que código de erro e mensagem devemos enviar para o usuário?

Vamos escrever um caso de teste unitário para esse comportamento, o caso de teste deve ser como o seguinte:

```
    it('should return 400 when an error occurs', () => {
      const request = {};
      const response = {
        send: sinon.spy(),
        status: sinon.stub()
      };

      response.status.withArgs(400).returns(response);
      Product.find = sinon.stub();
      Product.find.withArgs({}).rejects({ message: 'Error' });

      const productsController = new ProductsController(Product);

```

[59] http://mongoosejs.com/docs/queries.html
[60] https://en.wikipedia.org/wiki/Happy_path

```
14        return productsController.get(request, response)
15          .then(() => {
16            sinon.assert.calledWith(response.send, 'Error');
17          });
18      });
```

Devemos dar atenção a dois pontos nesse teste, primeiro é:

```
1  response.status.withArgs(400).returns(response);
```

Onde dizemos que: Quando a função *status* for chamada com o argumento *400* ela deve retornar o objeto *response*, isso por que a *API* do *express* concatena as chamadas de funções. O próximo ponto é:

```
1  Product.find.withArgs({}).rejects({message: 'Error'});
```

Aqui utilizamos o *stub* para rejeitar a *Promise* e simular uma consulta ao banco que retornou uma falha. Se executarmos os testes agora receberemos um erro, pois não implementamos ainda, então vamos implementar. Atualize a função *get* do *controller* de *products* adicionando um *catch* na busca, ele deve ficar assim:

```
1    get(req, res) {
2      return this.Product.find({})
3        .then(products => res.send(products))
4        .catch(err => res.status(400).send(err.message));
5    }
```

Aqui é dito que, quando ocorrer algum erro, o *status* da requisição será 400, usamos *res.status* que é uma função do *express* que adiciona o *statusCode* da resposta *HTTP*. Após isso enviamos a resposta adicionando a mensagem do erro como corpo utilizando a função *send* do objeto de resposta do *express*.

Agora basta executar os testes de unidade novamente e eles devem estar passando:

```
1  $ npm run test:unit
```

A resposta deve ser:

```
  Controllers: Products
    get() products
      ✓ should call send with a list of products
      ✓ should return 400 when an error occurs


  2 passing (223ms)
```

Nossa unidade está pronta para ser integrada com o resto da aplicação, faremos isso no próximo passo.

O passo *Refactor* do *TDD*

Lembram que nosso teste de integração está no passo *GREEN* do *TDD*? Ou seja, está com lógica suficiente para passar mas não está com a implementação real. Agora que o *controller* já está completo, integrando com o *model*, é o melhor momento para refatorar o resto dos componentes fazendo a integração com o *model* e *controller*.

Integração entre rota, *controller* e *model*

Nesse passo vamos refatorar nossa rota de *products* para que ela consiga criar o *controller* corretamente, passando o *model* como dependência. Altere o arquivo *src/routes/products.js* para que ele fique como o código a seguir:

```
import express from 'express';
  import ProductsController from '../controllers/products';
+ import Product from '../models/product';

  const router = express.Router();
- const productsController = new ProductsController();
+ const productsController = new ProductsController(Product);
  router.get('/', (req, res) => productsController.get(req, res));

  export default router;
```

A única mudança é que a nova instância do *controller* recebe o *model Product* por parâmetro. A integração parece estar pronta, vamos executar os testes de integração:

```
$ npm run test:integration
```

A saida sera como a seguinte:

```
Routes: Products
    GET /products
      1) should return a list of products


  0 passing (286ms)
  1 failing

  1) Routes: Products GET /products should return a list of products:
     Uncaught AssertionError: expected undefined to deeply equal { Object (name,\
 description, ...) }
      at Test.request.get.end (test/integration/routes/products_spec.js:41:34)
      at Test.assert (node_modules/supertest/lib/test.js:179:6)
      at Server.assert (node_modules/supertest/lib/test.js:131:12)
      at emitCloseNT (net.js:1549:8)
      at _combinedTickCallback (internal/process/next_tick.js:71:11)
      at process._tickCallback (internal/process/next_tick.js:98:9)
```

O teste falhou, e isso é esperado, pois agora utilizamos o *MongoDB* e vamos precisar criar um produto antes de executar o teste para que seja possível reproduzir o cenário que queremos.

Vamos adicionar o que precisamos no teste de integração da rota de productos, abra o arquivo *test/integration/routes/products_spec.js*. A primeira coisa é a resposta que esperamos do *MongoDB*. O *MongoDB* adiciona alguns campos aos documentos salvos que são *_v*, corresponde a versão do documento e *_id* que é o identificador único do documento, normalmente um *uuid.v4*[61].

```
const defaultProduct = {
      name: 'Default product',
      description: 'product description',
      price: 100
    };
+  const expectedProduct = {
+    __v: 0,
+    _id: '56cb91bdc3464f14678934ca',
+    name: 'Default product',
+    description: 'product description',
+    price: 100
+  };
+
```

Logo abaixo do *defaultProduct* adicionamos uma *constant* chamada *expectedProduct* correspondente ao produto que esperamos ser criado pelo *MongoDB*. Agora já possuímos o produto que queremos salvar que é *defaultProduct* e também o que esperamos de resposta do *MongoDB*.

[61] https://en.wikipedia.org/wiki/Universally_unique_identifier

Como estamos testando a rota products que retorna todos os produtos, precisamos ter produtos no banco de dados para poder validar o comportamento. Para isso iremos utilizar o *callback* do *Mocha* chamado *beforeEach*, que significa: antes de cada. Esse *callback* é executado pelo *Mocha* antes de cada caso de teste, então ele é perfeito para nosso cenário onde precisamos ter um produto disponível no banco antes de executar o teste.

Logo abaixo do código anterior adicione o seguinte código:

```
+ beforeEach(() => {
+   const product = new Product(defaultProduct);
+   product._id = '56cb91bdc3464f14678934ca';
+   return Product.remove({})
+     .then(() => product.save());
+ });
+
```

O que o código acima faz, é criar um novo produto utilizando os dados da *constant defaultProduct* e atribuir a nova instância do produto a *constant product*. Na linha seguinte a propriedade *product._id* do objeto criado pelo *mongoose* é sobrescrita por um *id* estático que geramos. Por padrão o *mongoose* gera um *uuid* para cada novo documento, mas no caso do teste precisamos saber qual é o *id* do documento que estamos salvando para poder comparar dentro do caso de teste, se utilizarmos o *uuid* gerado pelo *mongoose* o teste nunca conseguirá comparar o mesmo *id*. Dessa maneira sobrescrevemos por um *id* gerado por nós mesmos. Existem vários sites na *internet* para gerar *uuid*, aqui no livro por exemplo foi utilizado este: *uuid generator*[62].

Após a atribuição do *id* retornamos uma *Promise* que remove todos os produtos do banco de dados e depois salva o produto que criamos.

O próximo passo é garantir que iremos deixar o terreno limpo após executar o teste. Quando criamos testes que criam dados em banco de dados, escrevem arquivos em disco, ou seja, testes que podem deixar rastros para outros testes devemos limpar todo o estado e garantir que após a execução do teste não terá nenhum vestígio para os próximos. Para isso vamos adicionar também o *callback afterEach* que significa: Depois de cada, para garantir que o *MongoDB* ficará limpo, ou seja, sem dados. Para isso adicione o seguinte código logo abaixo do anterior:

```
+  afterEach(() => Product.remove({}));
```

O último passo é atualizar o caso de teste para que ele verifique o *expectedProduct* no lugar do *defaultProduct*:

[62] https://www.uuidgenerator.net/

```
    describe('GET /products', () => {
       it('should return a list of products', done => {

         request
         .get('/products')
         .end((err, res) => {
-          expect(res.body[0]).to.eql(defaultProduct);
+          expect(res.body).to.eql([expectedProduct]);
           done(err);
         });
       });
```

O código final do *products_spec.js* deve estar similar a este:

```
import Product from '../../../src/models/product';

describe('Routes: Products', () => {
  let request;

  before(()=> {
    return setupApp()
      .then(app => {
        request = supertest(app)
      });
  });

  const defaultProduct = {
    name: 'Default product',
    description: 'product description',
    price: 100
  };
  const expectedProduct = {
    __v: 0,
    _id: '56cb91bdc3464f14678934ca',
    name: 'Default product',
    description: 'product description',
    price: 100
  };

  beforeEach(() => {
    const product = new Product(defaultProduct);
    product._id = '56cb91bdc3464f14678934ca';
```

```
29      return Product.remove({})
30        .then(() => product.save());
31    });
32
33    afterEach(() => Product.remove({}));
34
35    describe('GET /products', () => {
36      it('should return a list of products', done => {
37
38        request
39          .get('/products')
40          .end((err, res) => {
41            expect(res.body).to.eql([expectedProduct]);
42            done(err);
43          });
44      });
45    });
46  });
```

Executando os testes de integração novamente:

```
$ npm run test:integration
```

Devemos ter a seguinte resposta:

```
  Routes: Products
    GET /products

      ✓ should return a list of products


  1 passing (307ms)
```

Nosso ciclo de *TDD* nos testes de integração está completo, refatoramos e adicionamos o comportamento esperado. Esse padrão onde começamos por testes de integração, depois criamos componentes internos como fizemos com *controllers* e *models* e utilizamos o teste de integração para válidar todo o comportamento, é conhecido como *outside-in*, termo esse que falaremos a seguir.

O código deste capitulo está disponível aqui[63].

[63] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step7