Construindo APIs testáveis com Node.js

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 2020-01-18

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:

#nodejs

Conteúdo

Introdução

Javascript é 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. Javascript é dona de uma fatia ainda única no mercado, pois é uma linguagem criada para browsers, para atualizar conteúdo dinamicamente, para ser não bloqueante, permitindo que ações sejam realizadas enquanto outras ainda estão sendo processadas. O contexto dos browsers contribuiu para que 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 Garrett 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 física, uma nova máquina virtual com uma nova instância da aplicação seria inicializada, permitindo a divisão da carga e dando origem a termos como micro-serviços. Nessa mesma época, 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 de 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 diferente de outras linguagens consolidadas no server-side como Java, que possui frameworks como SpringMVC para desenvolvimento web e 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 novos desenvolvedores, pois cada projeto possui uma arquitetura única o que torna complexa a escolha de um padrão

[1] http://stackoverflow.com/insights/survey/2017#technology
[2] http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

e a comparação das vantagens de cada um. Além disso, Javascript e 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
- Por que 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.

Sobre o autor

Waldemar Neto é engenheiro de software, contribuidor open-source, palestrante e criador de conteúdo. Começou sua carreira desenvolvendo servidores open source de MMRPGs em meados de 2004 e hoje possui mais de 15 anos de experiência na área de desenvolvimento. Passou por outras grandes empresas como Thoughtworks e POSSIBLE e atualmente é engenheiro de software na Atlassian (Sydney/Austrália).

Waldemar é apaixonado por metodologias ágeis e em seu tempo livre busca contribuir para a comunidade, principalmente criando conteúdo relacionado a microsserviços, apis e qualidade de software.

Agradecimentos

Quando comecei esse livro em 2017 minha ideia era criar um conteúdo único que trouxesse de forma simples e contínua conhecimento sobre Node.js, APIs e qualidade de software, algo que eu mesmo sentia muita falta na comunidade.

Logo nos primeiros capítulos publicados comecei a receber um ótimo feedback dos leitores, fiquei muito grato em saber que estava focando na coisa certa. Durante os 3 anos que o livro estava em desenvolvimento eu venho recebendo contribuições e feedback constante o que me ajudou a finalizar esse projeto.

Sou muito grato a comunidade brasileira do NodeBR e a os amigos e parentes que me ajudaram nessa trajetória. Um agradecimento especial aos meus ex colegas de Thoughtworks Rafael Gomes (Gomex), Glauco Vinicius, Lucas Lago, Juraci Vieira, William Calderipe, Mateus Tait e principalmente a minha namorada Andressa Costa pelas constantes revisões.

Introdução ao Node.js

Node.js não é uma linguagem de programação, 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 com que breaking changes eram 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 committers 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 committers decidiram fazer um fork do projeto e chamá-lo de IO.js com a intenção de prover releases mais rápidas e acompanhar as melhorias do Google V8.

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

O Google V8

O V8[4] é uma engine (motor) 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/
[4] https://v8.dev/

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:

```
1  const fs = require('fs');
2  let fileContent;
3  const someMath = 1+1;
4
5  try {
6    fileContent = fs.readFileSync('big-file.txt', 'utf-8');
7    console.log('file has been read');
8  } catch (err) {
9    console.log(err);
10 }
11
12 const text = `The sum is ${ someMath }`;
13
14 console.log(text);
```

A linha 14 do exemplo acima, com o `console.log`, imprimirá o resultado de `1+1` somente após 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:

```
1  const fs = require('fs');
2
3  const someMath = 1+1;
4
5  fs.readFile('big-file.txt', 'utf-8', function (err, content) {
6      if (err) {
7        return console.log(err)
8      }
9      console.log(content)
10 });
11
12 const text = `The response is ${ someMath }`;
13
14 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.

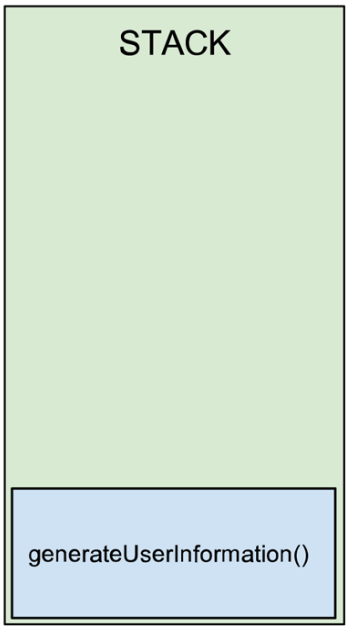
```
function generateBornDateFromAge(age) {
  return 2016 - age;
}

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:

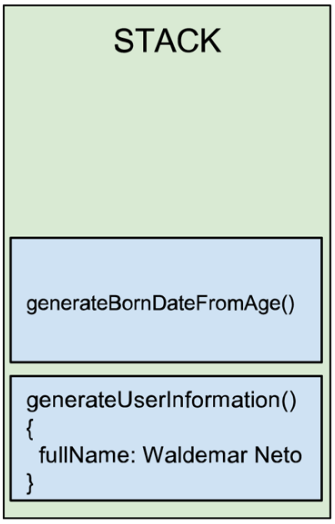
```
function generateBornDateFromAge(age) {
  return 2016 - age;
}

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:

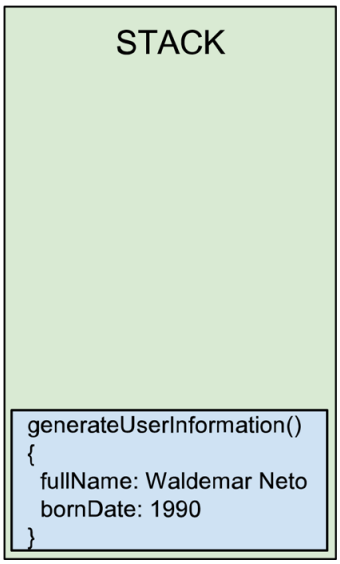
```
function generateBornDateFromAge(age) {
  return 2016 - age;
}

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`.

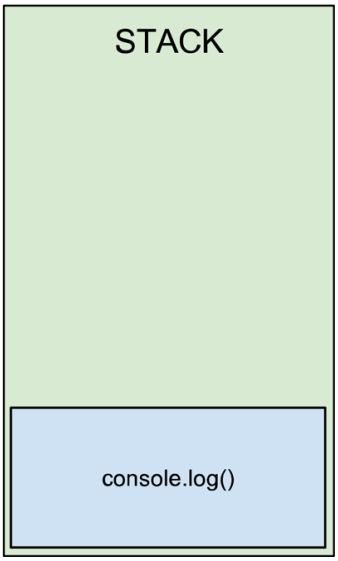
```
function generateBornDateFromAge(age) {
  return 2016 - age;
}

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.

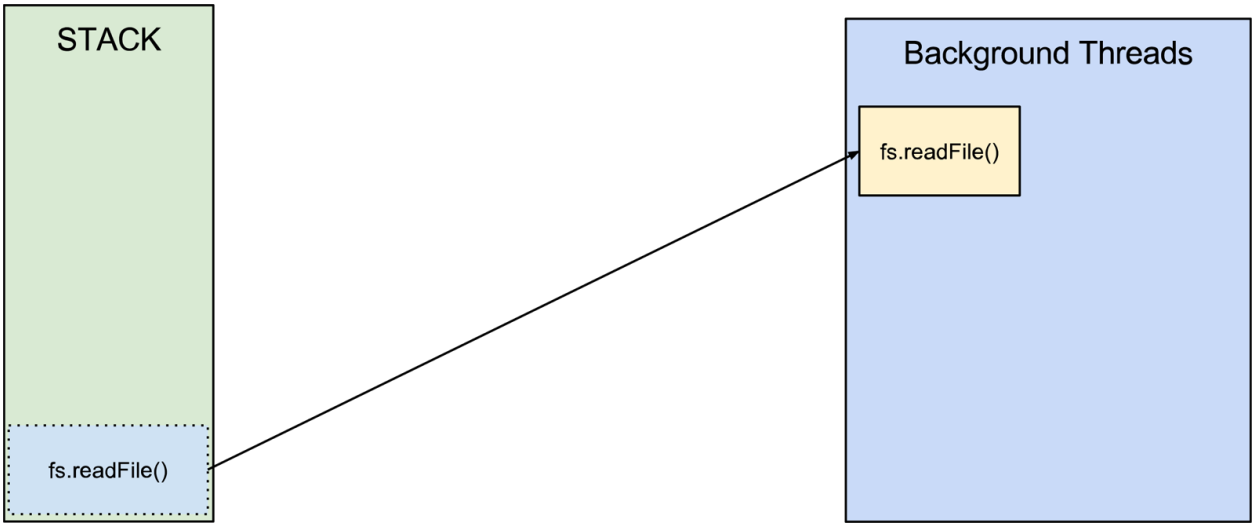
Multithreading

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.

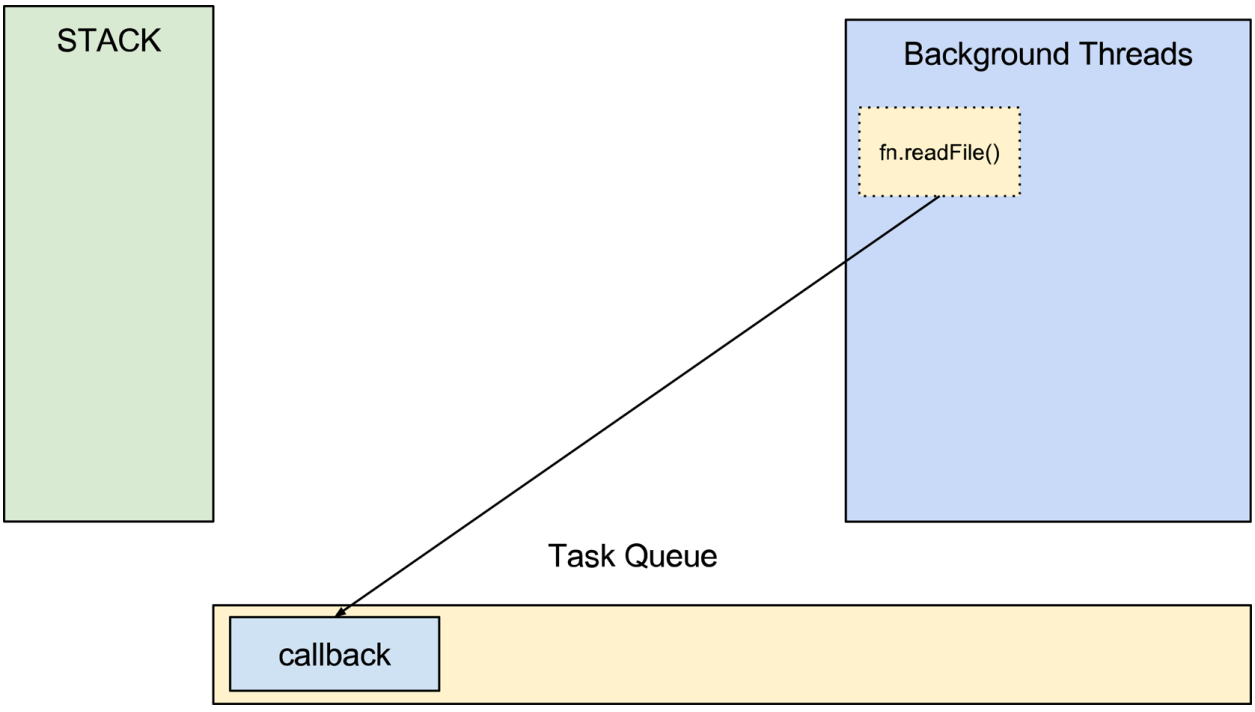
A libuv[5] é uma biblioteca open source multiplataforma escrita em C, criada inicialmente para o Node.js e hoje usada por diversos outros projetos como Julia[6] e Luvit[7].

O exemplo a seguir mostra uma função assíncrona sendo executada:

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



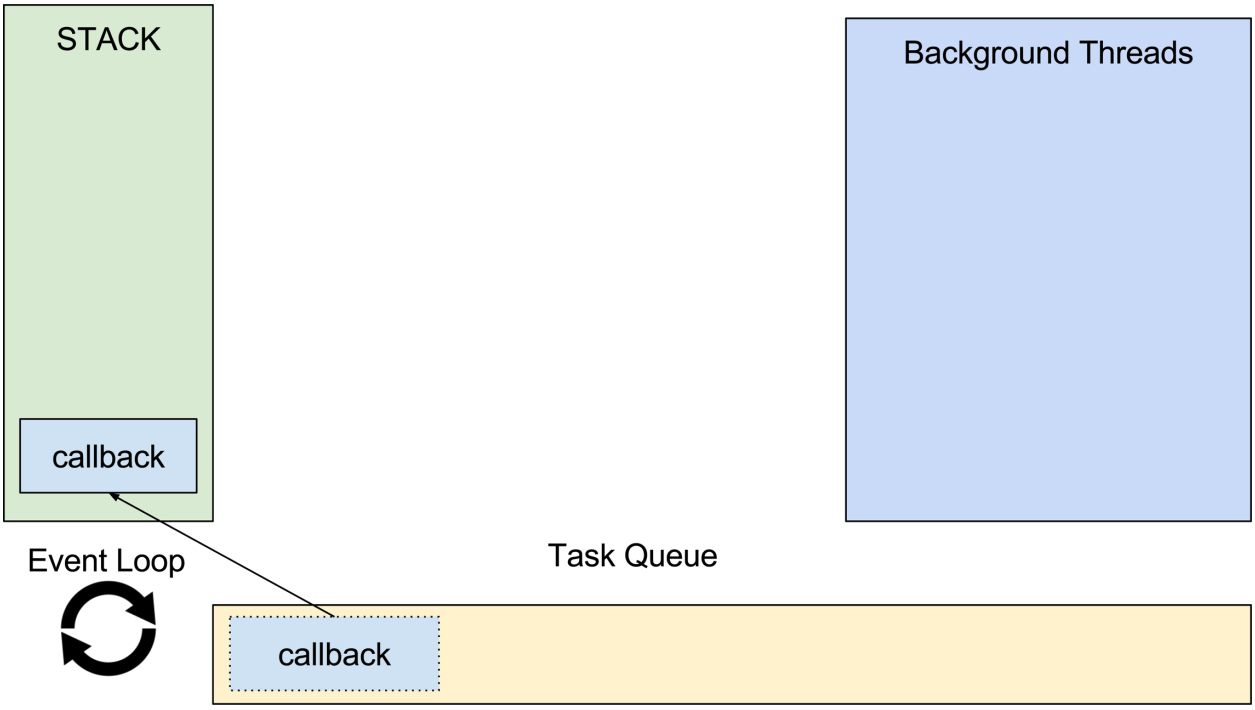
Nesse exemplo a função `readFile` do módulo de file system do Node.js é executada na stack e jogada para uma thread, a stack segue executando as próximas funções enquanto a função `readFile` está sendo administrada pela libuv em outra thread. Quando ela terminar, o callback sera adicionado a uma fila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

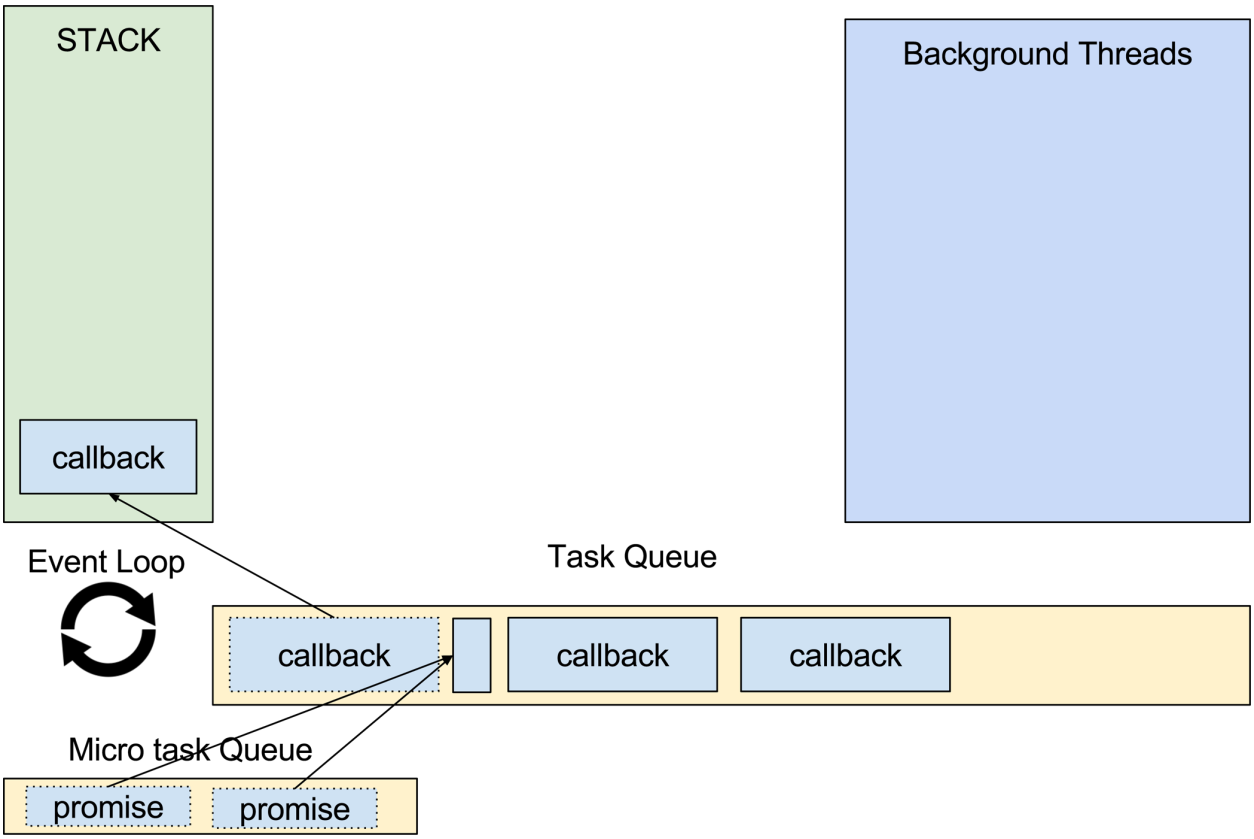
Alguns exemplos conhecidos de macro tasks são: setTimeout, I/O, setInterval. Segundo a especificação do WHATWG[8] somente uma macro task deve ser processada em um ciclo do Event Loop.

[8] 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[9]. 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[10], Gulp[11] e Brocoli[12] para gerenciar e automatizar tarefas e o Bower[13] 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[14] (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.

[9] https://babeljs.io/
[10] https://gruntjs.com/
[11] http://gulpjs.com/
[12] http://broccolijs.com/
[13] https://bower.io/
[14] 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/[15]. 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:

[15] 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@^7.7.4 @babel/core@^7.7.4 @babel/node@^7.7.4
```

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

```
$ npm install --save-dev @babel/preset-env@^7.7.4
```

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": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "node": true
        }
      }
    ]
  ]
}
```

Feito isso a aplicação já estará suportando 100% o ES6 e será possível utilizar as funcionalidades da versão. O *env* informa ao Babel pela ler o ambiente de desenvolvimento e adicionar as funcionalidades que estão faltando para a versão do Node.js que esta sendo utilizada. O código dessa etapa está disponivel neste link[16].

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[17], Hapi.js[18] e Koa.js[19]. Vamos optar pelo Express.js[20] por possuir um bom tempo de atividade, muito conteúdo na comunidade e é mantido pela Node Foundation[21].

O Express é um framework para desenvolvimento web para Node.js inspirado no Sinatra que é uma alternativa entre outros frameworks web para a linguagem ruby. Criado por TJ Holowaychuk o Express foi adquirido pela StrongLoop[22] 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:

```
$ npm install express@^4.14.0 body-parser@^1.15.2
```

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

[16] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step1
[17] http://sailsjs.com/
[18] https://hapijs.com/
[19] http://koajs.com/
[20] https://expressjs.com/
[21] https://nodejs.org/en/foundation/
[22] https://strongloop.com/

existe o modulo body-parser[23] 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:

```
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.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});
```

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": {
    "build": "babel src --out-dir dist",
    "start": "npm run build && node dist/server.js",
    "start:dev": "babel-node src/server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Build O Comando build utiliza o Babel para transpilar o código que foi escrito em Javascript suportado pelo ambiente, por exemplo se voce está utilizando Node na versão 12 e utilizando imports

[23] https://github.com/expressjs/body-parser

(ES6 Modules) o Babel vai transformar essa funcionalidade em algo suportado pela versão 12. Note que o código depois de transpilado será salvo no diretório *dist*. Esse código que deve ser utilizado em produção.

Start

O start é indicado para utilizar em produção, ele vai executar o comando build antes e depois iniciar a aplicação a partir do código transpilado.

Start dev

A diferença do start e do start:dev e que o start:dev utilizada o babel-node que transpila e inicia a aplicação ao mesmo tempo, vamo adicinar mais coisas a esse comando logo.

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

```
1 $ npm start
```

Agora a aplicação estará disponível em http://localhost:3000/. O código dessa etapa está disponivel neste link[24].

Entendendo o Middleware pattern

> Note esta etapa não faz parte do código da API que estamos construindo, o código é somente exemplo de implementação.

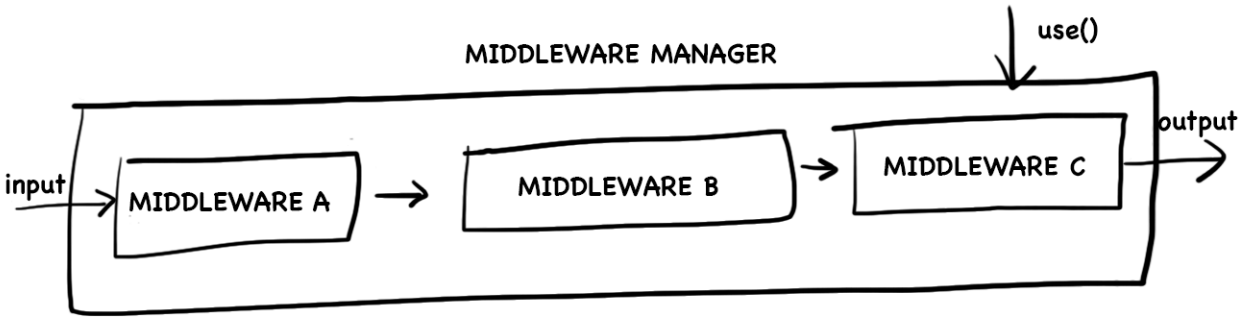
O padrão de Middleware implementado pelo express já é bem conhecido e tem sido usado por desenvolvedores em outras linguagens há muitos anos. Podemos dizer que se trata de uma implementação do padrão intercepting filter pattern[25] do chain of responsibility[26].

A implementação representa um pipeline de processamento onde handlers, units e filters são funções. Essas funções são conectadas criando uma sequência de processamento assíncrona que permite pré-processamento, processamento e pós-processamento de qualquer tipo de dado.

Uma das principais vantagens desse pattern é a facilidade de adicionar plugins de maneira não intrusiva.

O diagrama abaixo representa a implementação do Middleware pattern:

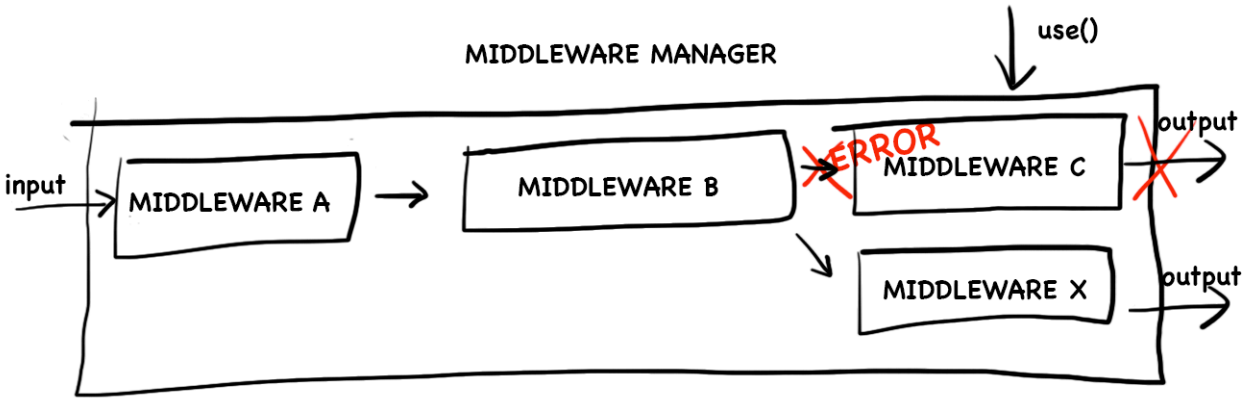
[24] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step2
[25] http://www.oracle.com/technetwork/java/interceptingfilter-142169.html
[26] http://java.dzone.com/articles/design-patterns-uncovered-chain-of-responsibility



O primeiro componente que devemos observar no diagrama acima é o Middleware Manager, ele é responsável por organizar e executar as funções. Alguns dos detalhes mais importantes dessa implementação são:

Novos middlewares podem ser invocados usando a função ‘use() (o nome não precisa ser estritamente use, aqui estamos usando o express como base). Geralmente novos middlewares são adicionados ao final do pipeline, mas essa não é uma regra obrigatória. Quando um novo dado é recebido para processamento, o middleware registrado é invocado em um fluxo de execução assíncrono. Cada unidade no pipeline recebe o resultado da anterior como input. Cada pedaço do middleware pode decidir parar o processamento simplesmente não chamando o callback, ou em caso de erro, passando o erro por callback. Normalmente erros disparam um fluxo diferente de processamento que é dedicado ao tratamento de erros.

O exemplo abaixo mostra um caminho de erro:



No express, por exemplo, o caminho padrão espera os parâmetros request, response e next, caso receba um quarto parâmetro, que normalmente é nomeado como error, ele vai buscar um caminho diferente.

Não há restrições de como os dados são processados ou propagados no pipeline. Algumas estratégias são:

- Incrementar os dados com propriedades ou funções.

- Substituir os dados com o resultado de algum tipo de processamento.
- Manter a imutabilidade dos dados sempre retornando uma cópia como resultado do processamento.

A implementação correta depende de como o Middleware Manager é implementado e do tipo de dados que serão processados no próprio middleware. Para saber mais sobre o pattern sugiro a leitura do livro Node.js Design Patterns[27].

Middlewares no Express

O exemplo a seguir mostra uma aplicação express simples, com uma rota que devolve um "Hello world" quando chamada:

```
1  const express = require('express');
2
3  const app = express();
4
5  app.get('/', function(req, res, next) {
6          console.log('route / called');
7    res.send('Hello World!');
8  });
9
10 app.listen(3000, () => {
11         console.log('app is running');
12 });
```

Agora vamos adicionar uma mensagem no console que deve aparecer antes da mensagem da rota:

```
1  const express = require('express');
2
3  const app = express();
4
5  app.use((req, res, next) => {
6    console.log('will run before any route');
7    next();
8  });
9
10 app.get('/', function(req, res, next) {
11         console.log('route / called');
12   res.send('Hello World!');
```

[27] https://www.packtpub.com/web-development/nodejs-design-patterns-second-edition

```
13 });
14
15 app.listen(3000, () => {
16         console.log('app is running');
17 });
```

Middlewares são apenas funções que recebem os parâmetros requisição (`req`), resposta (`res`) e próximo (`next`), executam alguma lógica e chamam o próximo `middleware` chamando `next`. No exemplo acima chamamos o use passando uma função que será o middleware, ela mostra a mensagem no console e depois chama o `next()`.

Se executarmos esse código e acessarmos a rota / a saída no terminal será:

```
1 app is running
2 will run before any route
3 route / called
```

Ok! Mas como eu sabia que iria executar antes? Como vimos anteriormente no middleware pattern, o middleware manager executa uma sequência de middlewares, então a ordem do use interfere na execução, por exemplo, se invertermos a ordem, como no código abaixo:

```
1  const express = require('express');
2
3  const app = express();
4
5
6  app.get('/', function(req, res, next) {
7          console.log('route / called');
8    res.send('Hello World!');
9  });
10
11 app.use((req, res, next) => {
12         console.log('will run before any route');
13         next();
14 });
15
16 app.listen(3000, () => {
17         console.log('app is running');
18 });
```

A saida será:

```
app is running
route / called
```

Dessa vez o nosso middleware não foi chamado, isso acontece porque a rota chama a função `res.send()` invés de `next()`, ou seja, ela quebra a sequência de middlewares.

Também é possível usar middlewares em rotas específicas, como abaixo:

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

const app = express();

app.use('/users', (req, res, next) => {
        console.log('will run before users route');
        next();
});

app.get('/', function(req, res, next) {
        console.log('route / called');
  res.send('Hello World!');
});

app.get('/users', function(req, res, next) {
        console.log('route /users called');
        res.send('Hello World!');
});

app.listen(3000, () => {
        console.log('app is running');
});
```

Caso seja feita uma chamada para / a saída sera:

```
app is running
route / called
```

Já para /users veremos a seguinte saída no terminal:

```
app is running
will run before users route
route /users called
```

O express também possibilita ter caminhos diferentes em caso de erro:

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

const app = express();

app.use((req, res, next) => {
        console.log('will run before any route');
        next();
});

app.use((err, req, res, next) => {
        console.log('something goes wrong');
        res.status(500).send(err.message);
});

app.get('/', function(req, res, next) {
        console.log('route / called');
  res.send('Hello World!');
});


app.listen(3000, () => {
        console.log('app is running');
});
```

Não mudamos nada no código de exemplo, apenas adicionamos mais um middleware que recebe o parâmetro `err`. Executando o código teremos a seguinte saída:

```
app is running
will run before any route
route / called
```

Apenas o primeiro middleware foi chamado, o middleware de erro não. Vamos ver o que acontece quando passamos um erro para o next do primeiro middleware.

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

const app = express();

app.use((req, res, next) => {
        console.log('will run before any route');
        next(new Error('failed!'));
});

app.use((err, req, res, next) => {
        console.log('something goes wrong');
        res.status(500).send(err.message);
});

app.get('/', function(req, res, next) {
        console.log('route / called');
  res.send('Hello World!');
});


app.listen(3000, () => {
        console.log('app is running');
});
```

A saída será:

```
app is running
will run before any route
something goes wrong
```

Essas são as formas mais comuns de utilizar middlewares, mas suas combinações são infinitas. Ao decorrer do livro utilizaremos middlewares para casos distintos.

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[28] 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.

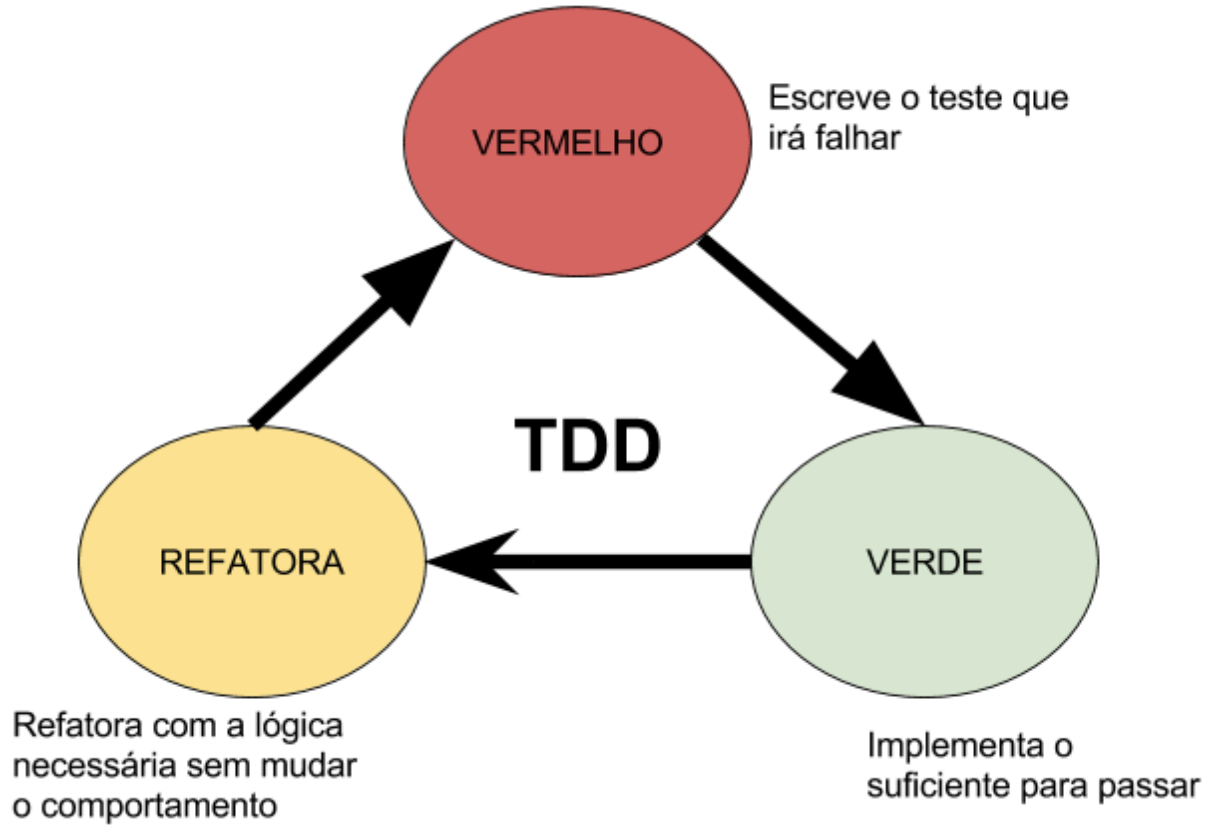
Green

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.

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

Refactor

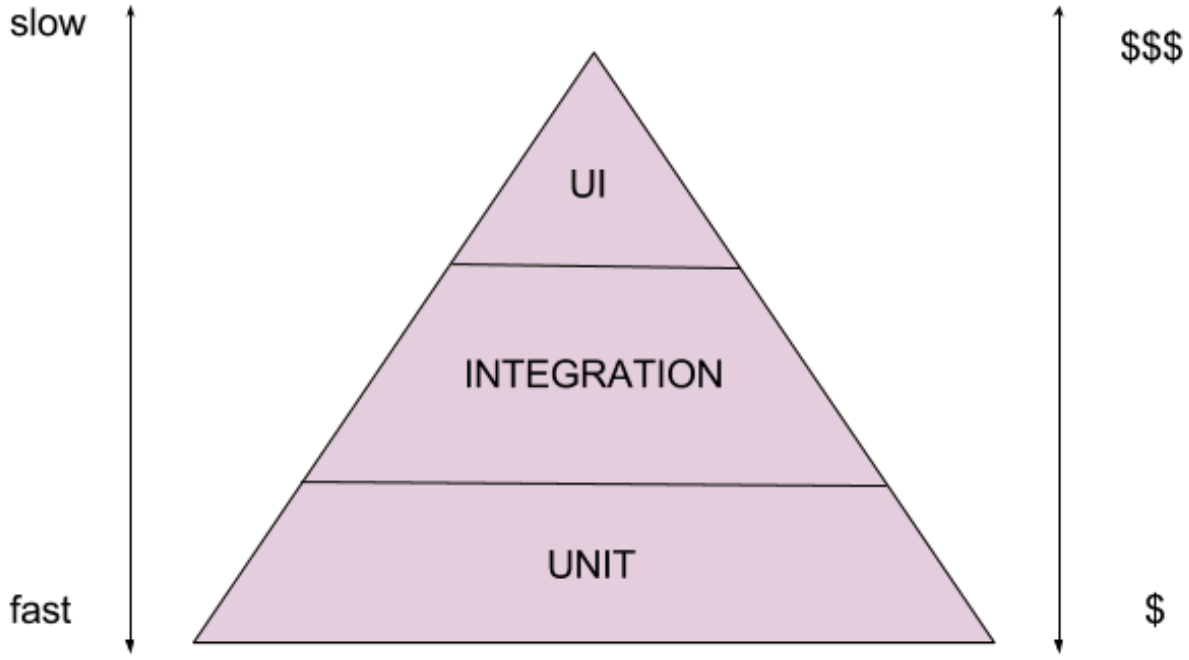
É onde se aplica a lógica necessária. Como o teste já foi validado nos passos anteriores, o refactor garantirá que a funcionalidade está sendo implementada corretamente. Nesse momento devem ser removidos os dados estáticos e todos itens adicionadas apenas para forçar o teste a passar, em seguida deve 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[29]. 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.

[29] 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[30] 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.

[30] 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:

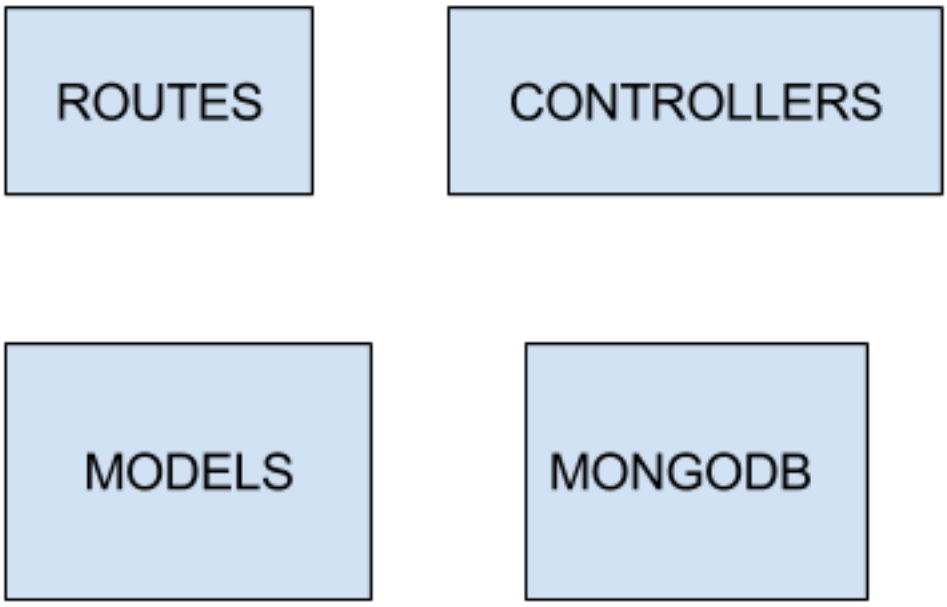
Para testar unitariamente é necessário isolar a classe Billing da sua dependência, a classe Orders, como na imagem a seguir:

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.

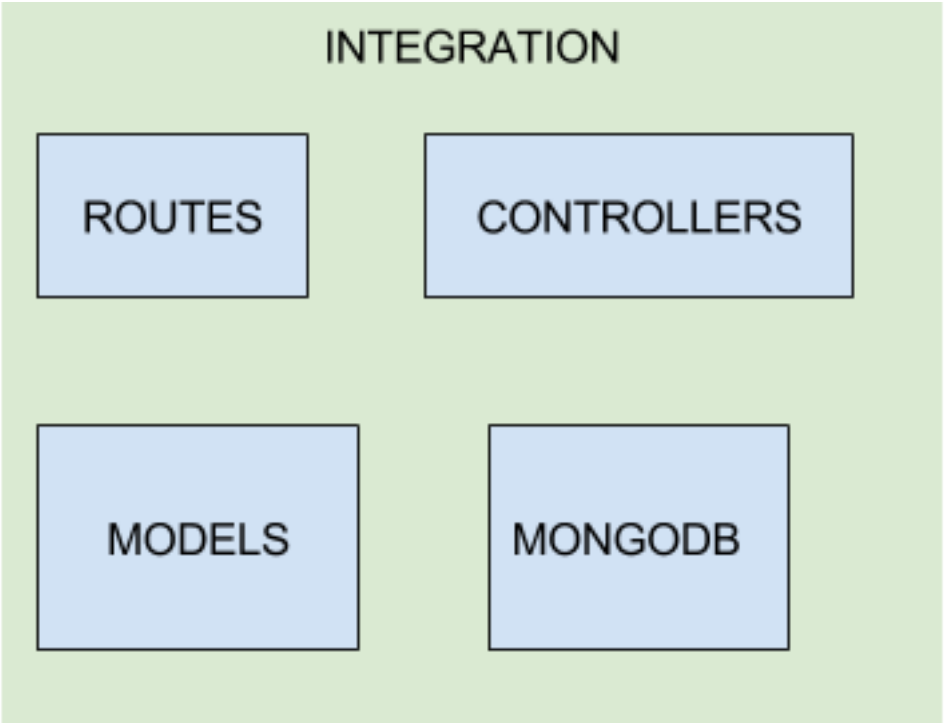
Testes de integração (Integration tests)

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.

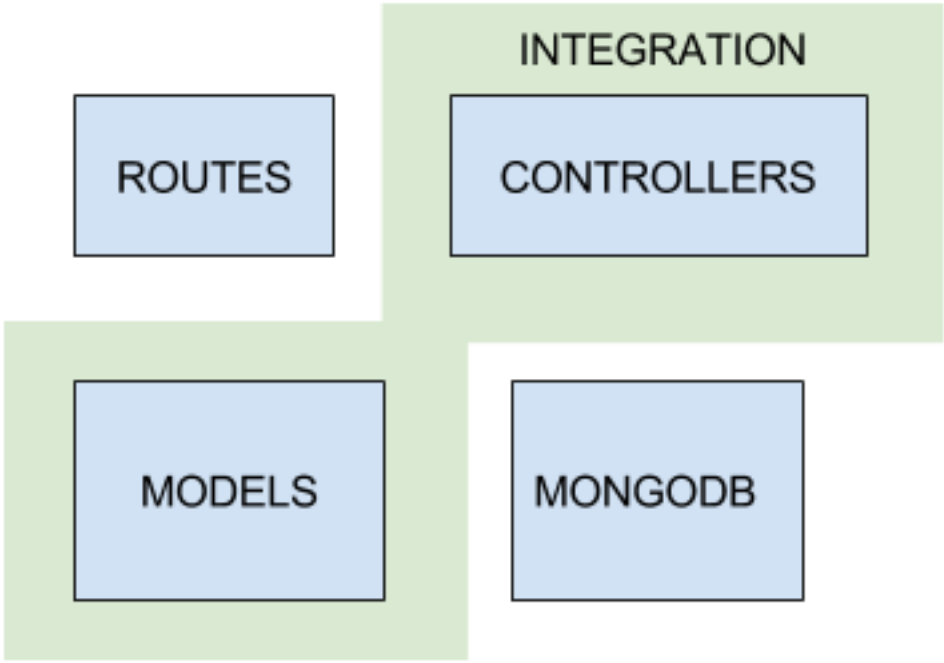
No exemplo a seguir temos uma arquitetura comum de aplicações Node.js e desejamos testar a integração entre as rotas, controllers, models e banco de dados:



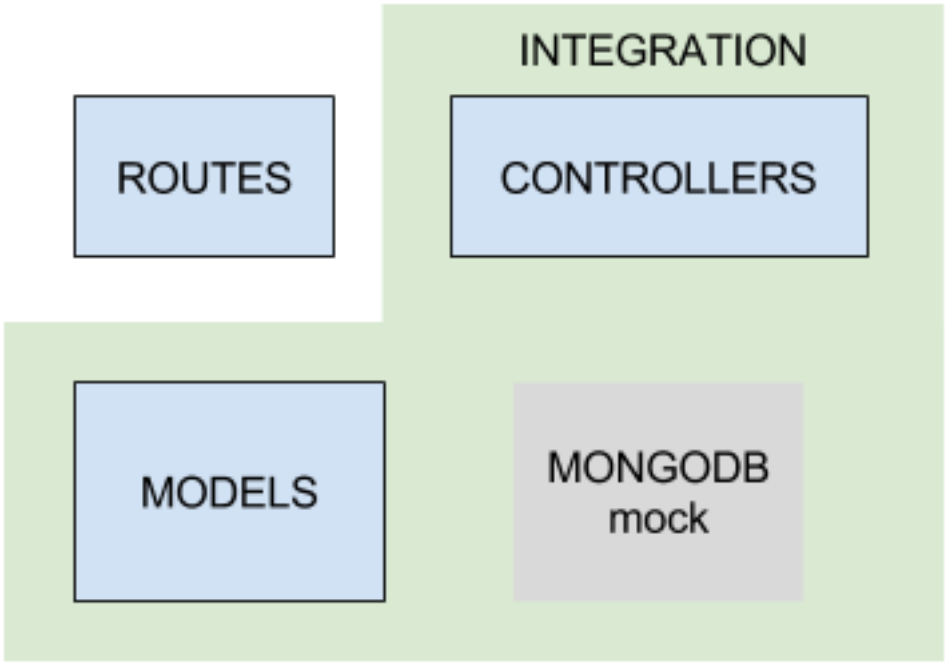
Nossa integração pode ser desde a rota até salvar no banco de dados (nesse caso, MongoDB), dessa maneira é possível validar todo o fluxo até o dado ser salvo no banco, como na imagem a seguir:



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.

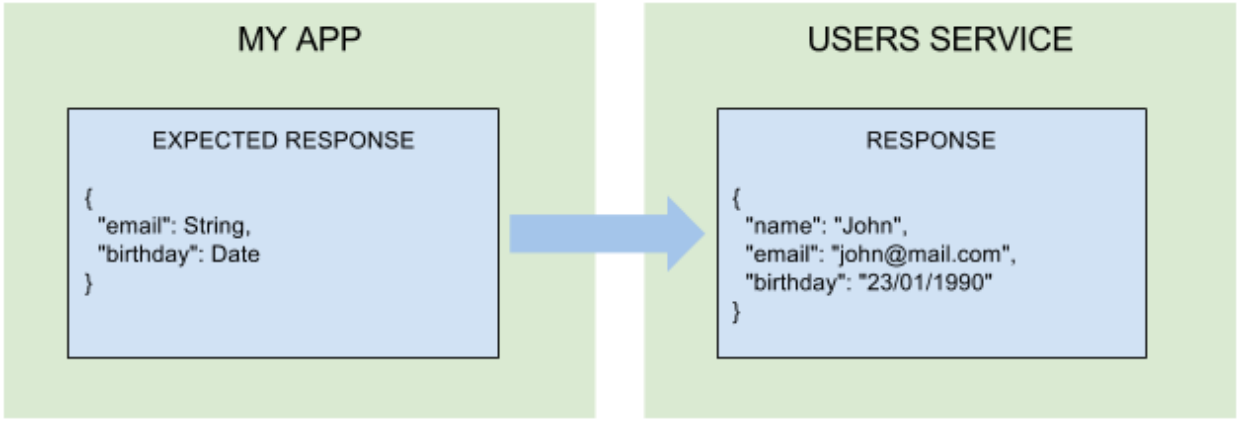


Teste de integração de contrato (Integration contract tests)

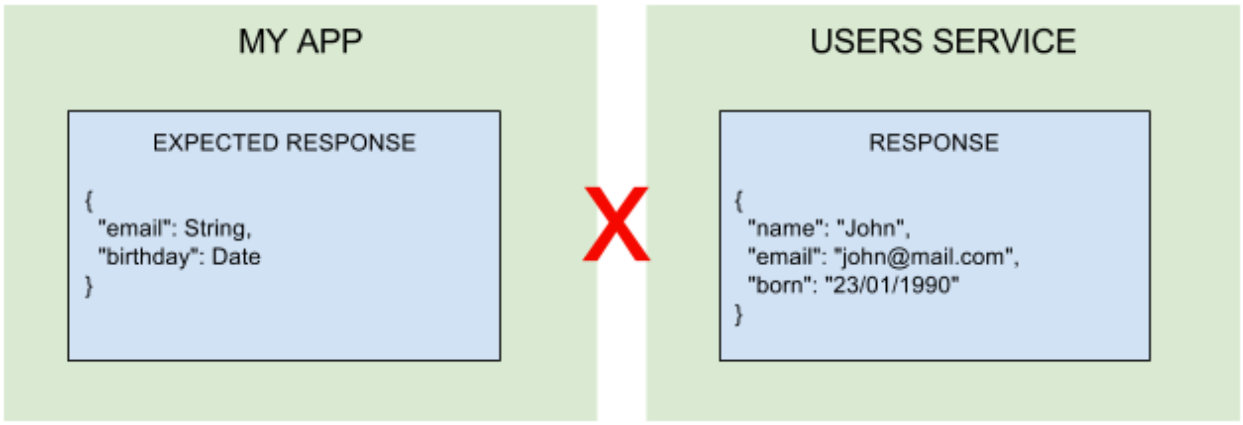
Testes de contrato ganharam muita força devido ao crescimento das APIs e dos micro serviços. Normalmente, quando testamos a nossa aplicação, mesmo com o teste de integração, tendemos a não usar os serviços externos e sim um substituto que devolve a resposta esperada. Isso por que serviços externos podem 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[31] 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

[31] 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[32] 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:

[32] 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:

```
describe('UsersController getAll()', () => {
  it('should database findAll with correct parameters', () => {
    const findAll = sinon.spy(Database, 'findAll');

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

    sinon.assert.calledWith(findAll, 'users');
    findAll.restore();
  });
});
```

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[33] e o Karma[34].

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[35] e o assert[36].

[33] https://mochajs.org/
[34] https://karma-runner.github.io/1.0/index.html
[35] http://chaijs.com/
[36] 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[37] e o TestDoubleJS[38] já para emular servidores existe o supertest[39].

[37] http://sinonjs.org/
[38] https://github.com/testdouble/testdouble.js
[39] 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@^6.2.2 chai@^4.2.0 supertest@^4.0.2
```

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 './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
├── src
|   └── app.js
|   └── server.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 @babel/register
--require test/integration/helpers.js
--reporter spec
--slow 5000
```

No primeiro require configuramos o Mocha com Babel, ja o segundo 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[40]. Reporters definem o estilo da saída do teste no terminal.
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:

[40] 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.js* 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
├── src
|   └── app.js
|   └── server.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/int\
egration/**/*_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, descri\
ption, ...) }
```

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[41].

[41] 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[42], 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*.

[42] https://github.com

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

Dentro do diretório source

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

```
├── src
|   ├── app.js
|   ├── server.js
|   ├── controllers
|   ├── middlewares
|   ├── models
|   └── 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
    ├── server.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
└── 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
    ├── server.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:

```
Import ProductsController from './src/controllers/products';
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
├── src
|   ├── app.js
|   ├── server.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[43]

[43] 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[44], 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 @babel/register
2  --require test/unit/helpers.js
3  --reporter spec
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/**/*_spec.js"
```

Para testar se o comando esta funcionando basta executar:

[44] 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@^7.5.0
```

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:

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

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

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

      const productsController = new ProductsController();
      productsController.get(request, response);

      expect(response.send.called).to.be.true;
      expect(response.send.calledWith(defaultProduct)).to.be.true;
    });
  });
});
```

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[45]

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

Configurando o MongoDB como banco de dados

Nesse livro o banco de dados escolhido foi o MongdoDB, principalmente pela simplicidade de ingração e vasto suporte opensource.

Introdução ao MongoDB

Até o ano de 2009 o mundo dos banco de dados era dominado por bancos RDMS[46] (Relational Database Management System) baseados em SQL (Structured Query Language). Esse tipo de banco de dados baseado em um modelo relacional foi introduzido em 1970 e é utilizado até hoje. Com o surgimento da Cloud e Big Data[47] os bancos relacionais começaram a enfrentar os desafios de escalar devido ao design relacional. Há 50 anos, quando os bancos de dados relacionais foram criados, o desafio que era armazenar dados, armazenamento era caro e lento.

Nos dias de hoje o armazenamento de dados é extremamente barato e não é mais visto como um problema, o desafio agora é está no processamento de dados. Empresas como a NASA com o projeto de monitoramento climático e de aquecimento global precisam processar e armazenar petabytes de dados. Pela natureza relacional, bancos RDMS não suportam bem esses casos, é difícil escalá-los horizontalmente já que os dados ficam em tabelas diferentes e são ligados por chaves.

Em 2009 o MongoDB surgiu, trazendo um design não relacional focado em prover escala e performance. O MongoDB armazena os dados de forma isolada possibilitando escalar horizontalmente de forma simples e com alta performance.

O MongoDB[48] segue um padrão chamado NoSQL. NoSQL não utilizada linguagem SQL e é livre de dados relacionais. Não existe um padrão fixo de NoSQL mas a maioria dos bancos de dados possui características similares como:

- Não possuem um schema fixo
- Tendem a serem escaláveis e distribuídos por padrão
- Utilizam diferentes estratégias para armazenar os dados de forma otimizada para escala e performance diferente da conhecida tabela, colunas e linhas dos bancos de dados RDMS.

Alguns paradigmas NoSQL são:

[46] https://en.wikipedia.org/wiki/Relational_database
[47] https://en.wikipedia.org/wiki/Big_data
[48] https://www.mongodb.com/

- Bancos em gráfico (Graph Databases)
- Armazenamento chave/valor (Key/Value store)
- Armazenamento por documentos (Document store)

O MongoDB segue o paradigma de armazenamento de documentos e conta com três itens chave: Document, Field e Collection

- Document: Similar a linha (row) de um banco de dados relacional
- Field: Similar a coluna (column) de um banco de dados relacional
- Collection: Similar a tabela (table) de um banco de dados relacional

Configurando o banco de dados com Mongoose

Para integrar nossa aplicação com o MongoDB vamos utilizar o Mongoose[49] 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:

```
$ npm install mongoose@^5.7.13
```

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:

```
├── src
|   ├── app.js
|   ├── server.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:

[49] http://mongoosejs.com/

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

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.

Logo abaixo vamos adicionar um método para fechar a conexão com o banco de dados, essa é uma boa prática que facilita muito nos casos de teste, depois de executar os testes fechamos o banco de dados e garantimos que nada esta executando em nossa aplicação.

```
const close = () => mongoose.connection.close();
```

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

```
export default {
  connect,
  connection: mongoose.connection
}
```

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

```
import mongoose from "mongoose";

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

const connect = () =>
  mongoose.connect(mongodbUrl, {
    useNewUrlParser: true,
    useUnifiedTopology: true
  });

export default {
  connect,
  close
};
```

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);
+  app.database = database;
+
+  return app;
+};
```

Acima criamos uma função nomeada `configureExpress` que terá a tarefa de configurar o express e retornar uma nova instância de aplicação configurada. Também adicionamos a instância do banco de dados, database, para ser parte da aplicação express dessa maneira e possivel abrir e fechar o banco de dados utilizando a app do express.

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 retornar a aplicação configurada para ser utilizada.

```
+ export default async() => {
+  const app = configureExpress();
+  await app.database.connect();
+
+  return app;
+};
```

No bloco acima exportamos uma função que retorna uma Promise esse passo é necesario pois 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 retornamos a instância da aplicação.

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

```
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);
  app.database = database;

  return app;
};

export default async () => {
  const app = configureExpress();
  await app.database.connect();

  return app;
};
```

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 './app';
+ import setupApp from './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}`);
-});
+(async () => {
+  try {
+    const app = await setupApp();
+    const server = app.listen(port, () =>
+      console.info(`app running on port ${port}`)
+    );
+
+    const exitSignals = ["SIGINT", "SIGTERM", "SIGQUIT"];
+    exitSignals.map(sig =>
+      process.on(sig, () =>
+        server.close(err => {
+          if (err) {
+            console.error(err);
+            process.exit(1);
+          }
+          app.database.connection.close(function() {
+            console.info("Database connection closed!");
+            process.exit(0);
+          });
+        })
+      )
+    );
+  } 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.error*, 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[50] de boas práticas para desenvolvimento de software moderno.

Graceful shutdown

O graceful shutdown é muito importante principalmente no momento em que a aplicação precisa escalar e rodar na nuvem onde normalmente ela sera executada dentro de um container. Orquestradores de containers como Kubernetes mandam comandos para a aplicação. A aplicação é responsavel por receber esse comando e administrar sua finalização antes de desligar, isso siginifica que a aplicação deve finalizar as conexões abertas, fechar a conexão com o banco de dados e então desligar. No código acima isso é feito utilizando adicionando event listeners a os sinais *SIGINT*, *SIGTERM*, *SIGQUIT* utilizando o *process.on(sig)*. Dessa maneira se a aplicação receber um dos sinais listados ela vai primeiro fechar o servidor express chamando o método close, *server.close()*, nesse momento o express vai fechar as conexões abertas e logo após fechar a conxão com o banco de dados, *connection.close()* e finalizar a aplicação com sucesso. Caso um erro ocorra para fechar a conexão do express o processo vai finalizar com 1, ou seja, falha. Desta maneira quem está finalizando o processo vai poder checar a saida (0) para sucesso ou (1) para falha para saber se a aplicação foi desligada com sucesso sem impactar nenhum cliente ou o banco de dados.

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)
```

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:

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

```
-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;
+  let app;
+
+  before(async () => {
+    app = await setupApp();
+    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. E também definimos o `let app` para armazenar a instancia da app do express, isso sera necessario apos executar todos os testes para fechar o banco de dados.

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[51].

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

[51] https://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:326:35)
      at Connection.<anonymous> (node_modules/mongodb-core/lib/connection/pool.js:27\
0:12)
      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 ultimo passo é adicionar o metódo `after` do Mocha que é chamado após todos os testes:

```
  after(async () => await app.database.connection.close());
```

Aqui garantimos que estamos fechando a conexão com o banco de dados após todos os testes, dessa maneira o Mocha vai executar os testes e logo após a execução vai fechar o banco e fechar o processo no terminal.

O código desta etapa esta disponível aqui[52].

[52] 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[53] trabalhava em uma linguagem de programação chamada Smalltalk[54], 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[55]. 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[56] (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[57], do Martin Fowler, que conta a história do MVC ao longo dos anos.

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.

[53] https://en.wikipedia.org/wiki/Trygve_Reenskaug
[54] https://en.wikipedia.org/wiki/Smalltalk
[55] https://martinfowler.com/eaaDev/uiArchs.html
[56] https://addyosmani.com/resources/essentialjsdesignpatterns/book/#observerpatternjavascript
[57] https://martinfowler.com/eaaDev/uiArchs.html

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[58] em como o MVC é implementado, muitas vezes omitindo o C, ou seja, não utilizando controllers, isso pois as 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[59] 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

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

[59] https://www.quora.com/What-is-two-way-data-binding-in-Angular

responsável por escutar e emitir eventos, o que atribui muita responsabilidade para os controllers, assim quebrando o princípio de responsabilidade única[60].

Em 2015 com o surgimento do React e frameworks como Flux, o MVC no front-end perdeu força[61] 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[62] a implementação de um controller para administrar as requisições e conversar com o model é obrigatório. Já no Express.js[63] 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.

[60] http://www.oodesign.com/single-responsibility-principle.html
[61] https://medium.freecodecamp.com/is-mvc-dead-for-the-frontend-35b4d1fe39ec#.u1yvsd5lt
[62] http://sailsjs.com/
[63] 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[64]:

> 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:

```
import mongoose from 'mongoose';

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

export default Product;
```

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

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:

```
1  //src/routes/products.js
2
3  import mongoose from 'mongoose';
4
5  console.log(mongoose.models);
```

O `console.log` mostrará:

```
1  { Product:
2     { [Function: model]
3  ...
```

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:

```
1  function Mongoose() {
2    this.connections = [];
3    this.plugins = [];
4    this.models = {};
5    this.modelSchemas = {};
6    // default global options
7    this.options = {
8      pluralization: true
9    };
10   var conn = this.createConnection(); // default connection
11   conn.models = this.models;
12 }
```

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 return a list of products', async() => {
      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);
+
+      await productsController.get(request, response);
+
+      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('Controllers: Products', () => {
  const defaultProduct = [
    {
      name: 'Default product',
      description: 'product description',
      price: 100
    }
  ];

  describe('get() products', () => {
    it('should return a list of products', async () => {
      const request = {};
      const response = {
        send: sinon.spy()
      };

      Product.find = sinon.stub();
      Product.find.withArgs({}).resolves(defaultProduct);

      const productsController = new ProductsController(Product);

      await productsController.get(request, response);

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

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:

```
class ProductsController {
+  constructor(Product) {
+    this.Product = Product;
+  };
```

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:

```
   get(req, res) {
-    return res.send([{
-      name: 'Default product',
-      description: 'product description',
-      price: 100
-    }])
+     const products = await this.Product.find({});
+     res.send(products);
    }
  }
```

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[65] irá retornar uma lista de objetos, então o que estásendo 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:

[65] http://mongoosejs.com/docs/queries.html

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


  1 passing (217ms)
```

Testando casos de erro

Até agora apenas testamos o happy path[66] (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', async () => {
      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);

      await productsController.get(request, response);

      sinon.assert.calledWith(response.send, 'Error');
    });
```

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

```
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:

[66] https://en.wikipedia.org/wiki/Happy_path

```
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:

```
  async get(req, res) {
    try {
      const products = await this.Product.find({});
      res.send(products);
    } catch (err) {
      res.status(400).send(err.message);
    }
  }
```

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:

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

A resposta deve ser:

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


  2 passing (13ms)
```

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, des\
cription, ...) }
      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[67].

```
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.

[67] 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(async() => {
+    await Product.deleteMany();
+
+    const product = new Product(defaultProduct);
+    product._id = '56cb91bdc3464f14678934ca';
+    return await 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[68].

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(async() => await Product.deleteMany());
```

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

[68] 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;
  let app;

  before(async () => {
    app = await setupApp();
    request = supertest(app);
  });

  after(async () => await app.database.connection.close());

  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(async() => {
    await Product.deleteMany();

```

```
30      const product = new Product(defaultProduct);
31      product._id = '56cb91bdc3464f14678934ca';
32      return await product.save();
33    });
34
35    afterEach(async() => await Product.deleteMany());
36
37    describe('GET /products', () => {
38      it('should return a list of products', done => {
39        request.get('/products').end((err, res) => {
40          expect(res.body).to.eql([expectedProduct]);
41          done(err);
42        });
43      });
44    });
45  });
```

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[69].

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

Behaviour Driven Development - BDD

Como vimos até agora, o ponto forte do TDD é garantir que o código possuirá testes automatizados e também guiar o desenvolvimento para um código desacoplado e legível.

Quando desenvolvemos guiados por testes utilizando TDD praticamos o chamado inside-out (em português, de dentro para fora) ou seja, começamos a testar pelas unidades e módulos internos e deixamos para fazer a integração dessas partes depois. Começar de dentro para fora leva a dúvidas, como: Como saber quando acabou? Será que está testado o suficiente? Será que isso deve ser testado? Ok, Está testado! mas será que quando integrar vai funcionar? Assim é muito fácil perder de vista o objetivo do que está sendo desenvolvido.

A fraqueza do TDD é induzir os desenvolvedores a testarem o código da maneira que eles entendem ao invés de testar se o código cumpre o que foi requerido pelo negócio.

Para contornar esse problema nasceu o BDD, Behaviour Driven Development (desenvolvimento guiado por comportamento). O lançamento do livro "The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends"[70] escrito por David Chelimsky com o apoio do Dan North, que também ajudou na escrita de ferramentas como Rspec, framework de testes para Ruby, e o Cucumber, uma ferramenta de automatização de testes, foi um dos responsáveis pela disseminação do BDD.

Como o BDD funciona

Quando um analista de negócio (também conhecido como BA que vem de Business Analyst) escreve um caso de uso ele detalha como o sistema deve se comportar. Quando ele escreve estórias de usuário[71] ele descreve critérios de aceitação que definem os comportamentos esperados do sistema.

O exemplo abaixo demonstra uma estória escrita por um BA para a criação de produtos em uma pagina de admin:

```
Feature: Product management
Scenario: User adds a new product
  Given I go to the new products page
  When I fill the form
  And I press submit
  Then I should be redirected to a page showing the created product
```

[70] https://pragprog.com/book/achbd/the-rspec-book
[71] http://www.agilemodeling.com/artifacts/userStory.htm

A principal ideia no exemplo acima é quebrar o cenário em três partes:

GIVEN (dado): Descreve o estado inicial, antes de começar o comportamento que está sendo especificado.

WHEN (quando): Descreve o comportamento esperado.

THEN (então): Descreve as mudanças esperadas baseadas no comportamento.

O exemplo anterior utiliza o Cucumber que é uma das ferramentas mais utilizadas para implementação de BDD. A grande vantagem do Cucumber é transformar especificações como a que vimos acima em casos de teste executáveis, permitindo que desenvolvimento, negócio e marketing trabalhem juntos validando se uma determinada funcionalidade está sendo implementada como esperado. Esse tipo de ferramenta é utilizada para business facing tests[72], testes que não serão validados por programadores. Com uma ferramenta que disponibiliza a validação dos comportamentos é possível que clientes, usuários, BA ou qualquer pessoa envolvida verifique se o comportamento esperado está sendo contemplado, quebrando a barreira de comunicação entre desenvolvedores e não desenvolvedores.

O GivenWhenThen[73] e o BDD não se prendem a uma ferramenta ou padrão de uso, pode-se utilizar os ensinamentos adotando o que for possível para melhorar a qualidade do que estamos desenvolvendo. Mesmo não trabalhando em um projeto ágil, onde não existam estórias de usuário nem Cucumber, é possível aplicar técnicas do BDD.

O outside-in

O principal foco do BDD é garantir que o comportamento descrito está sendo contemplado no desenvolvimento e para isso ele trabalha de fora para dentro.

Dan North no artigo What's in a Story[74] diz que:

O behaviour-driven development (BDD), é uma metodologia outside-in. Ela começa por fora, identificando as necessidades do negócio e depois parte para a definição das funcionalidades para alcançar essas necessidades. Cada funcionalidade é capturada como uma estória, na qual é definido o escopo da funcionalidade com o seu critério de aceitação.

No artigo é possível entender que o BDD funciona de fora para dentro em nível gerencial, mas isso também se aplica ao desenvolvimento da funcionalidade. Quando desenvolvemos uma funcionalidade seguindo a metodologia clássica do TDD tendemos a começar desenvolvendo as unidades do sistema, e depois fazer a integração entre elas, essa forma de desenvolver isola camadas e ofusca a visão do objetivo principal, o que realmente é importante para o usuário. É difícil saber se algo está sendo testado o suficiente e se vai funcionar quando integrado com as outras partes do sistema.

[72] https://martinfowler.com/bliki/BusinessFacingTest.html
[73] https://martinfowler.com/bliki/GivenWhenThen.html
[74] https://dannorth.net/whats-in-a-story/

Aplicando BDD e TDD é possível unir os benefícios do desenvolvimento guiado por testes com o valor do desenvolvimento guiado por comportamento, vamos falar sobre isso no próximo capítulo.

BDD com Mocha e Chai

Mocha e Chai são os módulos que utilizamos no decorrer do livro, ambos baseados no Rspec do Ruby. No exemplo a seguir veremos como utilizar as técnicas do BDD para guiar o desenvolvimento.

Exemplo com Mocha:

```
describe('Products Management', () => {
  describe('when adding products', () => {
    it('should save a product into the database', () => {
    });
  });
});
```

Como vimos no capítulo onde configuramos os testes, utilizamos o describe do Mocha para criar uma suíte de testes. No exemplo acima adicionamos uma suíte principal, que descreve o cenário como um todo, a Products Management, essa suíte de testes vai tratar do gerenciamento de produtos da aplicação.

Seguindo o GivenWhenThen adicionamos uma nova suíte que descreve o comportamento esperado: when adding products, ou seja, quando adicionar produtos então devem acontecer as mudanças esperadas. Para o Then utilizamos o `it` do Mocha. O `it` se refere a um caso de teste, no exemplo acima estabelecemos que: "should save a product into the database" (deve salvar um produto no banco de dados). Tanto o Mocha quanto o Chai são híbridos e permitem aplicar técnicas de TDD e BDD o `it` por exemplo é inspirado no BDD.

Quando executarmos o teste a saída no terminal será:

```
  Products Management
    when adding products
      ✓ should save a product into the database


  1 passing
```

Note que não é necessário nenhum código para entender a intenção desse teste, a maneira como ele foi construído descreve a intenção.

O Mocha é um test runner, sua responsabilidade é executar os testes. Para fazer as verificações conhecidas no TDD como asserts (asserções), e no BDD como expectations(expectativas), o Mocha possibilita utilizar módulos externos, como o Chai.

O Chai é uma biblioteca que tem a responsabilidade de fazer asserts/expectations, ele foi criado seguindo os princípios de descrição do BDD mas também suporta o TDD.

O combo Mocha/Chai ficou conhecido pela forma como os dois se complementam e possibilitam a criação de testes legíveis. No exemplo abaixo adicionamos uma expectation utilizando o Chai:

```
1 describe('Products Management', () => {
2   describe('when adding products', () => {
3     it('should save a product into the database', () => {
4
5       expect(Product.save()).to.be.equal(true);
6     });
7   });
8 });
```

O expect faz parte da API do Chai, com ele é possível encadear expectativas. O exemplo acima dispensa explicações, basta traduzir livremente para o português e temos a descrição exata do que está sendo verificado: Espera que `Product.save()` seja igual a true (verdadeiro).

Operações de CRUD

Neste capítulo serão adicionadas as operações de CRUD (Create, Read, Update, Delete) possibilitando a alteração do recurso product. Serão aplicadas as técnicas de BDD que foram vistas anteriormente, aqui chamaremos os testes de alto nível de teste de integração end 2 end (de ponta a ponta) ou teste de aceitação. Estes testes irão validar se a funcionalidade foi implementada como esperado.

Busca por id

Um padrão bastante comum em APIs REST é a busca de um recurso pelo identificador (id), por exemplo *http://minha_api/products/56cb91bdc3464f14678934ca* deve retornar o produto correspondente ao id informado na rota. Nossa API ainda não suporta essa funcionalidade, então vamos implementá-la. Como esse será um novo endpoint precisamos começar pelo teste de integração end 2 end que fica no arquivo: *test/integration/routes/products_spec.js*

Iniciaremos permitindo o reuso de código, tornando-o mais genérico, permitindo assim a reutilização:

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

O `_id` foi extraido do `expectedProduct` para uma constant isso será útil para testar a rota a seguir.

Dentro do cenário de testes: GET /products adicionaremos o seguinte teste:

```
 context('when an id is specified', done => {
      it('should return 200 with one product', done => {

        request
          .get(`/products/${defaultId}`)
          .end((err, res) => {
            expect(res.statusCode).to.eql(200);
            expect(res.body).to.eql([expectedProduct]);
            done(err);
          });
      });
    });
```

Note que esse caso de teste é similar ao "should return a list of products", que está dentro do mesmo cenário. Adicionamos um `context`, pois mesmo sendo no /products agora o foco será o contexto de busca por id onde é feito um GET para "products/56cb91bdc3464f14678934ca", ou seja filtrando somente um produto.

Adicionado o teste, vamos executá-lo:

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

A saida deve ser a seguinte:

```
    GET /products
      ✓ should return a list of products
      when an id is specified
        1) should return 200 with one product


  1 passing (141ms)
  1 failing

  1) Routes: Products GET /products when an id is specified should return 200 with o\
ne product:

      Uncaught AssertionError: expected 404 to deeply equal 200
      + expected - actual

      -404
      +200
```

Quebrou! Agora já temos por onde começar. Era esperado 200, código http de sucesso, e recebemos 404, código http de não encontrado (NOT_FOUND), o que significa que ainda não existe a rota que está sendo requisitada, esse deve ser o primeiro passo a ser implementado.

Vamos alterar o arquivo de rotas de produtos: *src/routes/products.js*, adicionando a seguinte rota:

```
router.get('/', (req, res) => productsController.get(req, res));
+ router.get('/:id', (req, res) => productsController.getById(req, res));
```

Executando os testes novamente o erro deve ser outro, como o seguinte:

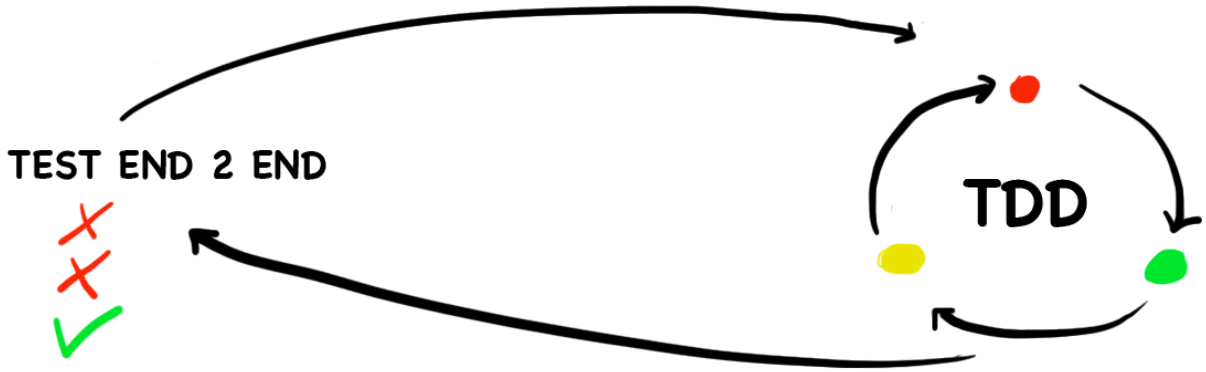
```
1) Routes: Products GET /products when an id is specified should return 200 with one\
 product:

      Uncaught AssertionError: expected 500 to deeply equal 200
      + expected - actual

      -500
      +200
```

500, Internal Server Error: Isso significa que a rota foi encontrada mas houve outro erro ao seguir a requisição, pois não temos mais nada implementado. O próximo passo é adicionar lógica para termos uma resposta.

O erro 500 aconteceu pois o método para buscar por id não existe no productsController, agora é o momento de criá-lo. Note que o teste end 2 end serve como um guia para garantir que estamos no caminho certo. Quando a rota foi criada não havia necessidade de testes pois a lógica era simples, já no controller o teste será necessário pois ele vai conter certa lógica. Nesse momento seguimos o ciclo do TDD. Algo como a imagem abaixo:



bdd end 2 end

O teste de ponta a ponta é o teste de aceitação, além de guiar o desenvolvimento ele também é responsável por validar se a funcionalidade que estamos desenvolvendo está ou não completa. Esse

teste não segue o fluxo do TDD, pois ele será executado inúmeras vezes até que passe, quando ele passar significa que tudo que é necessário para a funcionalidade estar completa foi desenvolvido.

Dentro desse grande teste de aceitação serão incluídos inúmeros outros testes, que podem ser de integração, de unidade e etc; esses testes sim seguirão o ciclo do TDD.

Os métodos que serão criados no controller seguirão o TDD como já vimos no livro. Vamos começar melhorando o reaproveitamento de código alterando o teste de unidade *test/unit/controllers/products_spec.js*:

```
+  const defaultRequest = {
+    params: {}
+  };

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

O objeto `request` foi movido para fora do teste e foi renomeado para `defaultRequest` para permitir sua reutilização por todos os casos de teste. Também adicionamos um objeto `params` dentro do `defaultRequest` para que fique similar ao objeto enviado pelo express.

A próxima alteração a ser feita é a requisição para o controller, como alteramos o nome de `request` para `defaultRequest` será necessário alterar a seguinte linha:

```
-      await productsController.get(request, response);
+      await productsController.get(defaultRequest, response);
```

Pronto, o teste deve estar assim:

```
  const defaultRequest = {
    params: {}
  };

  describe('get() products', () => {
    it('should return a list of products', async () => {
      const response = {
        send: sinon.spy()
      };

      Product.find = sinon.stub();
```

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

      const productsController = new ProductsController(Product);

      await productsController.get(defaultRequest, response);

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

Execute os testes de unidade, eles devem estar passando.

```
$ npm run test:unit


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


  2 passing
```

Testes verdes! Vamos criar o caso de teste para a busca por id, o teste ficará assim:

```
  describe('getById()', () => {
    it('should return one product', async () => {
      const fakeId = 'a-fake-id';
      const request = {
        params: {
          id: fakeId
        }
      };
      const response = {
        send: sinon.spy()
      };

      Product.find = sinon.stub();
      Product.find.withArgs({ _id: fakeId }).resolves(defaultProduct);

      const productsController = new ProductsController(Product);
      await productsController.getById(request, response);
```

```
18
19        sinon.assert.calledWith(response.send, defaultProduct);
20      });
21    });
```

O nome “should call send with one product” reflete o cenário que esperamos, ou seja, é esperado que o método `send` seja chamado com apenas um produto. Dentro do teste é criado uma constant chamada `fakeId` referente ao id do produto que será buscado. Logo após é criado um objeto `request` igual ao enviado pelo express nas requisições, quando um parâmetro é enviado o express adiciona ele dentro do objeto `params`, como no código acima onde adicionamos o id como parâmetro. A Próxima parte do código do teste que devemos dar atenção é esta:

```
1  Product.find.withArgs({ _id: fakeId }).resolves(defaultProduct);
```

Aqui é utilizado o stub do método `find` para adicionar um comportamento sempre que ele for chamado recebendo o parâmetro `_id` com o valor do `fakeId`. O `_id` é a chave primária do MongoDB então para fazer o filtro por id precisamos fazer uma busca pela chave `_id`. O método `withArgs` do stub do Sinon serve para adicionar um comportamento baseado em uma condição, no nosso caso quando o método `find` for chamado com o parâmetro `_id` com o valor do `fakeId` ele deve resolver uma Promise retornando o `defaultProduct`, simulando assim uma chamada ao banco de dados.

O método que será chamado é o `getById`, como no trecho abaixo:

```
1  await productsController.getById(request, response);
```

Vamos executar os testes de unidade:

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

Devemos receber o seguinte erro:

```
1  Controller: Products
2      get() products
3        ✓ should return a list of products
4        ✓ should return 400 when an error occurs
5      getById()
6        1) should call send with one product
7
8
9    2 passing (22ms)
10   1 failing
11
12   1) Controller: Products getById() should call send with one product:
13       TypeError: productsController.getById is not a function
```

O erro diz que o método `getById` não é uma , isso porque ainda não foi implementada a lógica, o stub que criamos não foi chamado e não retornou uma Promise. Vamos mudar o teste de unidade para o passo GREEN implementando o necessário para que o mesmo passe.

Devemos criar o método `getById` no controller de products que fica em: *src/controllers/products.js*. O código suficiente para o teste passar contém:

```
  async getById(req, res) {
    const response = await Promise.resolve([
      {
        __v: 0,
        _id: '56cb91bdc3464f14678934ca',
        name: 'Default product',
        description: 'product description',
        price: 100
      }
    ]);

    res.send(response);
  }
```

O trecho acima retorna uma Promise resolvida com um array contendo um produto fake, igual o que esperamos no caso de teste, e após o método `send` é chamado com esse produto.

Executando os testes de unidade:

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

A resposta deve ser:

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


  3 passing (20ms)
```

Teste unitário passando! Agora ele está no passo GREEN do TDD. Podemos partir para o REFACTOR.

Vamos alterar o método `getById` em *src/controllers/products.js* para ficar similar a este:

```
  async getById(req, res) {
    const {
      params: { id }
    } = req;

    try {
      const product = await this.Product.find({ _id: id });
      res.send(product);
    } catch (err) {
      res.status(400).send(err.message);
    }
  }
```

Na primeira linha extraímos o `id` do objeto params dentro de `req` e no método `find` do mongoose adicionamos um filtro por `id`.

Realizada a alteração basta executar os testes novamente, começando pelo teste de unidade:

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

A saida deve ser:

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


  3 passing (17ms)
```

Após os testes de unidade, devemos executar nosso teste end 2 end para validar:

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

A saida deve ser:

```
  Routes: Products
    GET /products
      ✓ should return a list of products
      when an id is specified
        ✓ should return 200 with one product


  2 passing (152ms)
```

O teste passou! Ou seja, nossa funcionalidade está implementada. Não se preocupe se ainda está meio confuso. Vamos aplicar esse conceito no decorrer do livro para a criação dos outros endpoints da API, aqui o objetivo é mostrar como o BDD e o TDD trabalham juntos e como a combinação dos dois ajuda no desenvolvimento criando um visão do que deve ser desenvolvido.

Criando um recurso

Nos passos anteriores trabalhamos nas buscas para listar todos os produtos e também filtrar por apenas um produto. Nesta etapa vamos trabalhar na criação de produtos na API.

Começaremos adicionando um teste de integração end 2 end no arquivo *test/integration/routes/products_spec.js* com o seguinte código:

```
   describe('POST /products', () => {
    context('when posting a product', () => {

      it('should return a new product with status code 201', done => {
        const customId = '56cb91bdc3464f14678934ba';
        const newProduct = Object.assign({},{ _id: customId, __v:0 }, defaultProduct\
);
        const expectedSavedProduct = {
          __v: 0,
          _id: customId,
          name: 'Default product',
          description: 'product description',
          price: 100
        };

        request
          .post('/products')
          .send(newProduct)
          .end((err, res) => {
            expect(res.statusCode).to.eql(201);
```

```
            expect(res.body).to.eql(expectedSavedProduct);
            done(err);
          });
      });
    });
  });
```

Usamos um novo `describe` pois separamos os cenários de testes por recursos da API, isso facilita a legibilidade e entendimento dos testes. O nosso teste deve criar um produto e retornar 201, com o produto criado. Note que é criado um `customId` e logo após sobrescrevemos o `id` do `defaultProduct` pelo `customId` usando `Object.assign`, para copiar o objeto e atribuir um novo valor ao `id`. Isso é necessário porque um novo produto será criado e ele precisa ter um `id` diferente do defaultProduct que já foi criado no `beforeEach`.

Em seguida criamos o `expectedSavedProduct`. Este é o objeto referente ao que esperamos que a rota de criação de produtos devolva no teste.

Na sequência, utilizamos o supertest para realizar um HTTP POST para a rota /products da API, enviando o objeto `newProduct`, anteriormente criado.

Quando a requisição terminar a resposta será validada:

```
expect(res.statusCode).to.eql(201);
expect(res.body).to.eql(expectedSavedProduct);
```

O teste vai verificar se a resposta da requisição é igual ao `expectedSavedProduct`, e o código http é igual a 201. Se sim, nosso produto foi criado com sucesso.

Executando os testes de integração, conforme:

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

Teremos a seguinte resposta:

```
 Routes: Products
    GET /products
      ✓ should return a list of products
      when an id is specified
        ✓ should return 200 with one product
    POST /products
      when posting a product
        1) should return a new product with status code 201


```

```
  2 passing (179ms)
  1 failing

  1) Routes: Products POST /products when posting a product should return a new prod\
uct with status code 201:

      Uncaught AssertionError: expected 404 to deeply equal 201
      + expected - actual

      -404
      +201
```

Já vimos esse cenário antes: esperávamos 200 e recebemos 404, ou seja, a rota não foi encontrada. Vamos adicioná-la no arquivo *src/routes/products.js*

```
router.get('/', (req, res) => productsController.get(req, res));
router.get('/:id', (req, res) => productsController.getById(req, res));
+ router.post('/', (req, res) => productsController.create(req, res));
```

Executando os testes novamente a saída deve ser:

```
      Uncaught AssertionError: expected 500 to deeply equal 201
      + expected - actual

      -500
      +201
```

Erro interno! É hora de implementar o controller.

Abra o teste de unidade em *test/unit/controllers/products_spec.js* e adicione o seguinte teste:

```
  describe('create() product', () => {
    it('should save a new product successfully', async () => {
      const requestWithBody = Object.assign(
        {},
        { body: defaultProduct[0] },
        defaultRequest
      );
      const response = {
        send: sinon.spy(),
        status: sinon.stub()
      };
      class fakeProduct {
```

```
13          save() {}
14        }
15
16        response.status.withArgs(201).returns(response);
17        sinon
18          .stub(fakeProduct.prototype, 'save')
19          .withArgs()
20          .resolves();
21
22        const productsController = new ProductsController(fakeProduct);
23
24        await productsController.create(requestWithBody, response);
25        sinon.assert.calledWith(response.send);
26      });
27    });
```

Para simular um objeto de request do express precisamos de um objeto que possua além das propriedades do `defaultRequest` também um body que contenha os dados enviados por post. Para isso é criado o `requestWithBody`, um novo objeto criado a partir dos dados padrão de request que usamos nos testes anteriores e adicionado um `body` com o `defaultProduct`. Dessa maneira possuímos uma requisição de post idêntica a enviada pelo express.

Os objetos response e `fakeProduct` seguem o mesmo padrão dos outros casos de teste. A única mudança é:

```
response.status.withArgs(201).returns(response);
```

Aqui definimos que `response.status` deve ser chamado com 201, ou seja, que o recurso foi criado com sucesso[75]; Para simular a ação de save no banco pelo model do mongoose adicionamos o seguinte stub:

```
sinon.stub(fakeProduct.prototype, 'save').withArgs().resolves();
```

Quando o método save do `fakeProduct` for chamado com qualquer argumento, ele vai retornar uma Promise resolvida.

Já temos os testes necessários e podemos rodar os testes de unidade:

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

A saída será:

[75] https://github.com/waldemarnt/http-status-codes#success-2xx

```
    Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      1) should save a new product successfully


  3 passing (22ms)
  1 failing

  1) Controller: Products create() product should save a new product successfully:
     TypeError: productsController.create is not a function
```

Ainda não criamos o método `create` no controller, esse será o nosso próximo passo. Vamos criar um método `create` no productsController:

```
async create(req, res) {

  return await Promise.resolve(res.send(req.body));
}
```

No teste verificamos se o `response.send` está sendo chamado com um produto criado, e esperamos por uma Promise. Essa é a menor implementação possível para atender ao teste. Ao executar os testes de unidade novamente devemos ter a seguinte resposta:

```
Controller: Products
  get() products
    ✓ should return a list of products
    ✓ should return 400 when an error occurs
  getById()
    ✓ should return one product
  create() product
    ✓ should save a new product successfully


  4 passing (26ms)
```

Estamos no estado GREEN dos testes de unidade. Vamos partir para o REFACTOR alterando o productsController, adicionando o seguinte:

```
  async create(req, res) {
    const product = new this.Product(req.body);

    await product.save();
    res.status(201).send(product);
  }
```

Após a alteração o teste deve estar passando com sucesso. Agora é necessário testar o caso de erro que é muito importante na hora de criar algum recurso.

Para testar um caso de erro precisamos que o método de criação de produto do ProductsController retorne um erro. Podemos criar este cenário tanto no teste de integração end 2 end quanto no teste de unidade. Como o teste de integração cobre a rota que recebe a resposta do controller e envia para o usuário, é indiferente a resposta que ele vai receber, independente de ser sucesso ou erro ela apenas será repassada. Para testarmos esse cenário com mais assertividade e mais controle dos componentes envolvidos vamos testar somente de forma unitária. Nos próximos capítulos vamos ver algumas maneiras de trabalhar com erros em testes de integração, mas neste momento vamos focar no nível unitário.

Adicione o seguinte teste no arquivo *test/unit/controllers/products_spec.js*:

```
    context('when an error occurs', () => {
      it('should return 422', async () => {
        const response = {
          send: sinon.spy(),
          status: sinon.stub()
        };

        class fakeProduct {
          save() {}
        }

        response.status.withArgs(422).returns(response);
        sinon
          .stub(fakeProduct.prototype, 'save')
          .withArgs()
          .rejects({ message: 'Error' });

        const productsController = new ProductsController(fakeProduct);

        await productsController.create(defaultRequest, response);
        sinon.assert.calledWith(response.status, 422);
      });
    });
```

Imagino que vocês já estejam treinados em escrever testes usando o Sinon. O caso de teste acima informa que esse teste deve retornar o código 422 quando acontecer um erro na criação de um novo produto.

Segundo a especificação do HTTP 1.1[76] o código 422 faz parte dos grupos de erro 4xx e significa "Unprocessable Entity" ou seja, a entidade não pode ser processada. Esse código de erro é utilizado para cenários onde a requisição foi recebida pelo servidor mas os dados não puderam ser validados. Um exemplo clássico é o caso do email, o usuário pode ter enviado os dados corretamente, mas o email é inválido. O servidor deve responder com 422, informando que recebeu os dados mas não conseguiu validar.

Para simular um caso de erro precisamos fazer com que o método de `save` do Mongoose retorne um erro. Como ele é uma Promise basta rejeitarmos o stub, como é feito aqui:

```
sinon.stub(fakeProduct.prototype, 'save').withArgs().rejects({ message: 'Error' });
```

Executando os testes de unidade:

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

A saida será:

```
  Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        1) should return 422


  4 passing (26ms)
  1 failing

  1) Controller: Products create() product when an error occurs should return 422:
     Error
```

Como já era esperado recebemos um erro, pois ainda não implementamos essa lógica. Vamos atualizar o método create no ProductsController e adicionar um catch para pegar o erro quando caso ele ocorra:

[76] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

```
  async create(req, res) {
    const product = new this.Product(req.body);
+    try {
       await product.save();
       res.status(201).send(product);
+    } catch (err) {
+      res.status(422).send(err.message);
+    }
  }
```

Executando os testes novamente, a saída deve ser:

```
  Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422


  5 passing (28ms)
```

Perfeito! Nossa rota de criação de produtos está pronta. Vamos nos certificar de que a implementação da funcionalidade está correta executando o teste end 2 end:

```
$ npm run test:integration

Routes: Products
    GET /products
      ✓ should return a list of products
      when an id is specified
        ✓ should return 200 with one product
    POST /products
      when posting a product
        ✓ should return a new product with status code 201


  3 passing (169ms)
```

Atualizando um recurso

Já é possível criar e listar produtos na nossa API, o próximo passo é a edição. Como de costume, vamos começar pelo teste end 2 end descrevendo o comportamento esperado dessa funcionalidade.

Adicione o seguinte cenário contendo um caso de teste no arquivo *test/integration/routes/products_-spec.js*

```
1   describe('PUT /products/:id', () => {
2     context('when editing a product', () => {
3       it('should update the product and return 200 as status code', done => {
4         const customProduct = {
5           name: 'Custom name'
6         };
7         const updatedProduct = Object.assign({}, customProduct, defaultProduct)
8
9         request
10          .put(`/products/${defaultId}`)
11          .send(updatedProduct)
12          .end((err, res) => {
13            expect(res.status).to.eql(200);
14            done(err);
15          });
16      });
17    });
18  });
```

Esse teste é muito similar ao teste de criação de produto, a única alteração é o verbo e a rota para a qual a requisição é feita, conforme o seguinte trecho de código:

```
1       .put(`/products/${defaultId}`)
```

Segundo o Rest[77], para a criação de um novo recurso utilizamos o verbo POST e para a atualização de um recurso devemos utilizar PUT.

O produto que será atualizado é o `defaultProduct`, este produto é criado antes de cada teste pelo `beforeEach`. Para não atualizar o `defaultProduct` diretamente vamos criar um objeto a partir dele usando `Object.assign`:

```
1   const updatedProduct = Object.assign({}, customProduct, defaultProduct)
```

Executando os testes de integração:

[77] http://restcookbook.com/HTTP%20Methods/put-vs-post/

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

A saída deve ser:

```
   Routes: Products
    GET /products
      ✓ should return a list of products
      when an id is specified
        ✓ should return 200 with one product
    POST /products
      when posting a product
        ✓ should return a new product with status code 201
    PUT /products/:id
      when editing a product
        1) should update the product and return 200 as status code


  3 passing (403ms)
  1 failing

  1) Routes: Products PUT /products/:id when editing a product should update the pro\
duct and return 200 as status code:

      Uncaught AssertionError: expected 404 to deeply equal 200
      + expected - actual

      -404
      +200
```

O teste retornar que esperava 200 (sucesso) mas recebeu 404 (não encontrado), como já esperávamos. Vamos alterar o arquivo *src/routes/products.js* e adicionar a seguinte rota:

```
+ router.put('/:id', (req, res) => productsController.update(req, res));
```

Executando os testes novamente a saída deve ser:

```
  1) Routes: Products PUT /products/:id when editing a product should update the pro\
duct and return 200 as status code:

      Uncaught AssertionError: expected 500 to deeply equal 200
      + expected - actual

      -500
      +200
```

É hora de descer para o nível de unidade para fazer a implementação no controller. Vamos adicionar o seguinte cenário de teste unitário em *test/unit/controllers/products_spec.js*

```
  describe('update() product', () => {
    it('should respond with 200 when the product has been updated', async () => {
      const fakeId = 'a-fake-id';
      const updatedProduct = {
        _id: fakeId,
        name: 'Updated product',
        description: 'Updated description',
        price: 150
      };
      const request = {
        params: {
          id: fakeId
        },
        body: updatedProduct
      };
      const response = {
        sendStatus: sinon.spy()
      };

      class fakeProduct {
        static updateOne() {}
      }

      const updateOneStub = sinon.stub(fakeProduct, 'updateOne');
      updateOneStub
        .withArgs({ _id: fakeId }, updatedProduct)
        .resolves(updatedProduct);

      const productsController = new ProductsController(fakeProduct);

      await productsController.update(request, response);
```

```
32        sinon.assert.calledWith(response.sendStatus, 200);
33      });
34    });
```

Este teste é maior, mas não há nada que já não tenhamos feito em outros testes. A chave para o update de um recurso é o `id` dele. Criamos uma constant chamada `fakeId` para poder reutilizá-lo em outras partes do teste, em seguida criamos um objeto chamado `updatedProduct` que representa o produto que será atualizado pelo Mongoose.

Para simular a requisição que será feita pelo express criamos um objeto com as mesmas chaves que o express envia na requisição, como podemos ver aqui:

```
1       const request = {
2         params: {
3           id: fakeId
4         },
5         body: updatedProduct
6       };
```

O método do Mongoose que utilizaremos para atualizar o recurso é o updateOne[78], segundo a documentação é necessário passar o `_id` e o objeto que queremos atualizar. Para testar isoladamente criamos um model fake, o `fakeProduct`, que possui o método a seguir:

```
1       class fakeProduct {
2         static updateOne() {}
3       }
```

Para adicionar o comportamento esperado no método `updateOne` vamos transformá-lo em um stub:

```
1       const updateOneStub = sinon.stub(fakeProduct, 'updateOne');
```

Depois, definimos que quando o stub for chamado com um objeto que contenha uma chave `_id` e um objeto igual ao `updatedProduct`, ele deve resolver uma Promise:

```
1 updateOneStub.withArgs({ _id: fakeId }, updatedProduct).resolves(updatedProduct);
```

Segundo a documentação do Mongoose, o método `updateOne` não retorna valor, por isso a Promise não irá retornar nada, somente será resolvida para dar sucesso.

Executando os testes de unidade:

[78] http://mongoosejs.com/docs/api.html#query_Query-updateOne

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

Vamos ter a seguinte saída:

```
  Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422
    update() product
      1) should respond with 200 when the product is updated


  5 passing (38ms)
  1 failing

  1) Controller: Products update() product should respond with 200 when the product \
is updated:
     TypeError: productsController.update is not a function
      at Context.it (test/unit/controllers/products_spec.js:154:
```

O teste retorna que o método `update` não existe, então vamos adicioná-lo com lógica suficiente apenas para que ele passe:

```
        async update(req, res) {
                res.sendStatus(200);
                return await Promise.resolve();
        }
```

No teste acima esperamos que o objeto `response` do express seja chamado com 200, o que garante que o produto foi atualizado. Essa é a implementação mínima para fazer o teste passar.

Executando os testes de unidade:

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

Devemos ter a seguinte saída:

```
 Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422
    update() product
      ✓ should respond with 200 when the product is updated


  6 passing (37ms)
```

Vamos à refatoração, o código do método `update` deve ficar assim:

```
  async update(req, res) {
    await this.Product.updateOne({ _id: req.params.id}, req.body);
    res.sendStatus(200);
  }
```

Vamos executar os testes de unidade:

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

A saida deve ser:

```
   Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422
    update() product
      ✓ should respond with 200 when the product is updated


  6 passing (27ms)
```

Também vamos executar os testes de integração end 2 end:

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

A saída deve ser:

```
  Routes: Products
    GET /products
      ✓ should return a list of products
      when an id is specified
        ✓ should return 200 with one product
    POST /products
      when posting a product
        ✓ should return a new product with status code 201
    PUT /products/:id
      when editing a product
        ✓ should update the product and return 200 as status code


  4 passing (184ms)
```

Todos os testes estão passando e a atualização de produtos está funcionando. O próximo passo é adicionar um teste para o caso de algum erro acontecer, similar ao que já foi feito na criação de produtos.

Vamos atualizar o teste *test/unit/controllers/products_spec.js* adicionando o seguinte caso de teste dentro do cenário update:

```
    context('when an error occurs', () => {
      it('should return 422', async() => {
        const fakeId = 'a-fake-id';
        const updatedProduct = {
          _id: fakeId,
          name: 'Updated product',
          description: 'Updated description',
          price: 150
        };
        const request = {
          params: {
            id: fakeId
          },
          body: updatedProduct
        };
```

```
        const response = {
          send: sinon.spy(),
          status: sinon.stub()
        };

        class fakeProduct {
          static updateOne() {}
        }

        const updateOneStub = sinon.stub(
          fakeProduct,
          'updateOne'
        );
        updateOneStub
          .withArgs({ _id: fakeId }, updatedProduct)
          .rejects({ message: 'Error' });
        response.status.withArgs(422).returns(response);

        const productsController = new ProductsController(fakeProduct);

        await productsController.update(request, response);
        sinon.assert.calledWith(response.send, 'Error');

        });
      });
```

Não há nada de novo comparado a o que foi feito no `create`, foi utilizada a mesma técnica de stub como podemos ver aqui:

```
updateOneStub.withArgs({ _id: fakeId }, updatedProduct).rejects({ message: 'Error' }\
);
```

Ao executar os testes de unidade:

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

Devemos ter a seguinte saída:

```
  Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422
    update() product
      ✓ should respond with 200 when the product is updated
      when an error occurs
        1) should return 422


  6 passing (26ms)
  1 failing

  1) Controller: Products update() product when an error occurs should return 422:
     Error
```

Como já esperávamos, o teste está falhando. Vamos atualizar método `update` do productsController adicionando o método método `catch`:

```
  async update(req, res) {
+    try {
      await this.Product.updateOne({ _id: req.params.id }, req.body);
      res.sendStatus(200);
+    } catch (err) {
+      res.status(422).send(err.message);
+    }
  }
```

Executando os testes novamente, a saída deve ser:

```
  Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422
    update() product
      ✓ should respond with 200 when the product is updated
      when an error occurs
        ✓ should return 422


  7 passing (46ms)
```

Note que não fizemos o processo GREEN, isso porque a implementação era clara e simples. Não é necessário escrever código por obrigação, o estágio de Green serve para ajudar e validar o teste.

Deletando um recurso

Já temos o C.R.U do CRUD, o último passo é o DELETE que vai permitir a remoção de recursos da API.

Como sempre, começamos pelo teste de integração end 2 end em *test/integration/routes/products_-spec.js*

```
describe('DELETE /products/:id', () => {
  context('when deleting a product', () => {
    it('should delete a product and return 204 as status code', done => {

      request
        .delete(`/products/${defaultId}`)
        .end((err, res) => {
          expect(res.status).to.eql(204);
          done(err);
        });
    });
  });
});
```

Dessa vez enviamos uma requisição do tipo DELETE:

```
      .delete(`/products/${defaultId}`)
```

Passando um id de produto. Segundo a especificação do HTTP/1.1[79] o método `delete` serve para deletar um recurso do servidor com base na url enviada. A documentação também diz que as respostas podem ser: 200 - a resposta contém um corpo, 202 se a ação não vai ser realizada agora (vai ocorrer em background) ou 204 quando não há retorno, somente a notificação de sucesso. Na maioria das APIs Rest é comum o uso do código 204. Ele é um código de sucesso utilizado para momentos onde é necessário notificar o sucesso, mas a resposta não vai ter dados.

Após adicionar o teste, vamos executar os testes:

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

A saída deve ser a seguinte:

```
    Routes: Products
    GET /products
      ✓ should return a list of products
      when an id is specified
        ✓ should return 200 with one product
    POST /products
      when posting a product
        ✓ should return a new product with status code 201
    PUT /products/:id
      when editing a product
        ✓ should update the product and return 200 as status code
    DELETE /products/:id
      when deleting a product
        1) should delete a product and return 204 as status code


  4 passing (221ms)
  1 failing

  1) Routes: Products DELETE /products/:id when deleting a product should delete a p\
roduct and return 204 as status code:

      Uncaught AssertionError: expected 404 to deeply equal 204
      + expected - actual

```

[79] https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

```
      -404
      +204
```

Como esperado, temos como retorno o código 404 (Not Found). Vamos adicionar a rota no arquivo *src/routes/products.js*.

```
+ router.delete('/:id', (req, res) => productsController.remove(req, res));
```

Executando os testes novamente o retorno deve ser:

```
 1) Routes: Products DELETE /products/:id when deleting a product should delete a pr\
oduct and return 204 as status code:

      Uncaught AssertionError: expected 500 to deeply equal 204
      + expected - actual

      -500
      +204
```

Agora é hora de adicionar os testes de unidade para o controller. No arquivo *test/unit/controllers/-products_spec.js* adicione o seguinte cenário de teste com o caso de teste a seguir:

```
  describe('delete() product', async() => {
    it('should respond with 204 when the product has been deleted', () => {
      const fakeId = 'a-fake-id';
      const request = {
        params: {
          id: fakeId
        }
      };
      const response = {
        sendStatus: sinon.spy()
      };

      class fakeProduct {
        static deleteOne() {}
      }

      const deleteOneStub = sinon.stub(fakeProduct, 'deleteOne');

      deleteOneStub.withArgs({ _id: fakeId }).resolves();

```

```
21        const productsController = new ProductsController(fakeProduct);
22
23        await productsController.remove(request, response);
24        sinon.assert.calledWith(response.sendStatus, 204);
25        });
26    });
```

O método utilizado para remover um produto é o `deleteOne`, segundo a documentação do Mongoose[80] ele recebe por parametro as condições para deletar o item, no nosso caso o id. Para simular esse cenário vamos criar o seguinte stub no código acima:

```
1         deleteOneStub.withArgs({ _id: fakeId }).resolves([1]);
```

Aqui informamos que quando o método `deleteOne` for chamado com um `_id` igual ao `fakeId` deve resolver uma Promise devolvendo um array com 1 elemento.

Executando os testes de unidade devemos ter a seguinte saída:

```
1     Controller: Products
2       get() products
3         ✓ should return a list of products
4         ✓ should return 400 when an error occurs
5       getById()
6         ✓ should return one product
7       create() product
8         ✓ should save a new product successfully
9         when an error occurs
10          ✓ should return 422
11      update() product
12        ✓ should respond with 200 when the product is updated
13        when an error occurs
14          ✓ should return 422
15      delete() product
16        1) should respond with 204 when the product is deleted
17
18
19    7 passing (46ms)
20    1 failing
21
22    1) Controller: Products delete() product should respond with 204 when the product \
23  is deleted:
24       TypeError: productsController.remove is not a function
25        at Context.it (test/unit/controllers/products_spec.js:220:33)
```

[80] https://mongoosejs.com/docs/api.html#model_Model.deleteOne

O método `remove` não foi encontrado, então vamos criá-lo no ProductsController, o código deve ficar assim:

```
        async remove(req, res) {
                res.sendStatus(204);
                return await Promise.resolve();
        }
```

Executando os testes de unidade, devemos ter a seguinte saída:

```
    Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422
    update() product
      ✓ should respond with 200 when the product is updated
      when an error occurs
        ✓ should return 422
    delete() product
      ✓ should respond with 204 when the product is deleted


  8 passing (32ms)
```

Agora no passo de refatoração adicionaremos a lógica real do método:

```
  async remove(req, res) {
    await this.Product.deleteOne({ _id: req.params.id });
    res.sendStatus(204);
  }
```

Ambos os testes devem seguir passando com sucesso.

O último passo é o tratamento dos possíveis erros. Nos testes de unidade vamos adicionar o seguinte teste dentro do cenário do método `remove`:

```
    context('when an error occurs', () => {
      it('should return 400', async() => {
        const fakeId = 'a-fake-id';
        const request = {
          params: {
            id: fakeId
          }
        };
        const response = {
          send: sinon.spy(),
          status: sinon.stub()
        };

        class fakeProduct {
          static deleteOne() {}
        }

        const deleteOneStub = sinon.stub(fakeProduct, 'deleteOne');

        deleteOneStub.withArgs({ _id: fakeId }).rejects({ message: 'Error' });
        response.status.withArgs(400).returns(response);

        const productsController = new ProductsController(fakeProduct);

        await productsController.remove(request, response);
        sinon.assert.calledWith(response.send, 'Error');
      });
    });
```

Executando os testes de unidade devemos ter a seguinte saída:

```
  Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422
    update() product
      ✓ should respond with 200 when the product is updated
```

```
      when an error occurs
        ✓ should return 422
    delete() product
      ✓ should respond with 204 when the product is deleted
      when an error occurs
        1) should return 400


  8 passing (35ms)
  1 failing

  1) Controller: Products delete() product when an error occurs should return 400:
     Error
```

A implementação ficará da seguinte maneira:

```
async remove(req, res) {
  try {
    await this.Product.deleteOne({ _id: req.params.id });
    res.sendStatus(204);
  } catch (err) {
    res.status(422).send(err.message);
  }
}
```

Note que agora retornamos 400 e não mais 422, 400 significa Bad Request e é um erro genérico, como o delete não recebe e nem valida dados não caberia utilizar o código 422.

Executando os testes de unidade agora a saída deve ser:

```
  Controller: Products
    get() products
      ✓ should return a list of products
      ✓ should return 400 when an error occurs
    getById()
      ✓ should return one product
    create() product
      ✓ should save a new product successfully
      when an error occurs
        ✓ should return 422
    update() product
      ✓ should respond with 200 when the product is updated
      when an error occurs
```

```
14          ✓ should return 422
15      delete() product
16        ✓ should respond with 204 when the product is deleted
17        when an error occurs
18          ✓ should return 400
19
20
21    9 passing (50ms)
```

Pronto! As operações de CRUD para o recurso de produtos estão prontas. O código dessa parte pode ser encontrado neste link[81].

[81] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step8

Configuração por ambiente

Antes de seguir adicionando funcionalidades a nossa API vamos fazer algumas melhorias. Como temos testes que nos garantem confiança no código, podemos melhorar o design do nosso código sempre que preciso, buscando sempre evoluir a arquitetura. Vamos começar pelo simples (sempre!), evitando complexidades antes que elas sejam realmente necessárias, e vamos evoluindo a arquitetura conforme a demanda. Nos próximos capítulos vamos adicionar autenticação e migrações de banco de dados, também teremos algumas configurações que precisarão ser feitas.

Essas modificações tornam necessário mover o nosso diretório de *config* para fora de *src*, assim será possível o reuso das configurações do ambiente; essa parte ficará mais clara durante a implementação.

Alterando a arquitetura

Vamos começar movendo o arquivo *database.js* para o raiz do diretório *src* e a pasta *src/config* para fora do diretório *src*, a estrutura deve ficar como a seguir:

```
├── config
├── src
|   ├── app.js
|   ├── controllers
|   ├── database.js
|   ├── models
|   ├── routes
|   └── server.js
└── test
```

Feito isso vamos executar os testes:

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

Os testes de unidade devem estar passando, agora executaremos os testes de integração:

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

Os testes de integração vão quebrar pois o arquivo de configuração não foi encontrado. Vamos fazer as atualizações necessárias para que o código passe a utilizar a configuração a partir do diretório correto.

Vamos alterar o arquivo *src/app.js* da seguinte maneira:

```
-import database from './config/database'
+import database from './database';
```

Ao executar os testes de integração novamente eles devem passar.

Configurações por ambiente

Provavelmente nossa aplicação irá rodar em vários ambientes, como desenvolvimento, homologação, produção e também teste. Muitas configurações são específicas por ambiente, como por exemplo o nome/url do banco de dados que vai ser diferente entre os ambientes, e isso deve ser fácil de ser configurado e versionado. Para nos ajudar nessa missão utilizaremos o módulo *node-config*[82].

Utilizando o módulo node-config

O módulo config suporta múltiplos ambientes, variáveis de ambiente e muitas outras formas de configuração, o que é muito útil na hora de criar aplicações que irão rodar na nuvem ou até mesmo em diferentes servidores. Para começar, vamos instalar o módulo config com o seguinte comando:

```
$ npm install config@1.29.4
```

Após a instalação precisamos fazer a configuração. No próprio repositorio no github https://github.com/lorenwest/no
config[83] temos uma vasta documentação de como configurar. Resumindo: será necessário criar um arquivo json para cada ambiente e neste arquivo adicionar as propriedades que queremos. Por padrão ele espera que os arquivos de configuração estejam dentro do diretório *config*, na raiz do projeto, por isso a mudança na arquitetura foi necessária.

Vamos começar criando uma configuração padrão para a nossa API, caso não tenha um arquivo específico para o ambiente o diretório *config* será utilizado. Vamos criar o arquivo *default.json* no diretório *config/*. Neste arquivo teremos a seguinte instrução:

```
{
  "database": {
    "mongoUrl": "mongodb://localhost:27017/shop"
  }
}
```

Essa é nossa configuração padrão, agora vamos criar uma configuração para teste, crie um arquivo *test.json* no diretório *config* com a seguinte instrução:

[82] https://github.com/lorenwest/node-config
[83] https://github.com/lorenwest/node-config

```
{
  "database": {
    "mongoUrl": "mongodb://localhost:27017/test"
  }
}
```

Nesse caso o mongo vai utilizar um banco de teste. O próximo passo é alterar a aplicação para fazer uso dessas configurações. Começamos importando o módulo config no *config/database.js*, como no trecho de código abaixo:

```
 import mongoose from 'mongoose';
+import config from 'config';
```

Em seguida, vamos alterar a linha onde definimos a constante `mongodbUrl` para que ela utilize o valor que vem do config:

```
-const mongodbUrl = process.env.MONGODB_URL || 'mongodb://localhost:27017/test';
+const mongodbUrl = config.get('database.mongoUrl');
```

Dessa maneira quem irá cuidar da configuração é o config, ele que vai ter a responsabilidade de entregar o `mongoUrl` baseado no ambiente em que a aplicação está sendo executada.

O módulo config utiliza a variável de ambiente `NODE_ENV` para saber qual arquivo carregar, como definimos no package.json o `NODE_ENV=test` para nossos testes, ele vai carregar o arquivo *test.json*.

```
 "test:integration": "NODE_ENV=test mocha --opts test/integration/mocha.opts test/in\
tegration/**/*_spec.js",
 "test:unit": "NODE_ENV=test mocha --opts test/unit/mocha.opts test/unit/**/*_spec.j\
s"
```

O módulo config trabalha de forma hierárquica sobrescrevendo os arquivos de ambiente, não é necessário colocar todas as variáveis em um novo json, somente as que serão alteradas para determinado ambiente, dessa forma ele usará o valor default para as que não foram alteradas.

Para finalizar vamos alterar o *package.json* adicionando o comando de test, para que seja possível executarmos ambos os testes de unidade e integração com apenas um comando:

```
-    "test": "echo \"Error: no test specified\" && exit 1",
+    "test": "npm run test:unit && npm run test:integration",
```

Para testar basta executar o seguinte comando:

```
$ npm test
```

Os testes de unidade e em seguida os testes de integração serão executados.

O código deste capitulo esta disponível em: passo 9[84]

[84] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step9

Usuários e autenticação

Antes de entrarmos na autenticação precisamos implementar a parte de usuários da nossa aplicação. Vamos começar criando tests, controller, model e a configuração das rotas para os usuários.

Neste capítulo não vou mostrar o ciclo de TDD passo-a-passo, pois já focamos nisso anteriormente, se surgir alguma dúvida lembre-se que é o mesmo processo que seguimos ao criar a parte de products.

Começaremos pelo controller, pelos testes unitários, crie um arquivo em *test/unit/controllers/users_-spec.js* com o seguinte código:

```
1  import UsersController from '../../../src/controllers/users';
2  import sinon from 'sinon';
3  import User from '../../../src/models/user';
4
5  describe('Controller: Users', () => {
6    const defaultUser = [
7      {
8        __v: 0,
9        _id: '56cb91bdc3464f14678934ca',
10       name: 'Default User',
11       email: 'user@mail.com',
12       password: 'password',
13       role: 'user'
14     }
15   ];
16
17   const defaultRequest = {
18     params: {}
19   };
20
21   describe('get() users', () => {
22     it('should return a list of users', async () => {
23       const response = {
24         send: sinon.spy()
25       };
26       User.find = sinon.stub();
27
28       User.find.withArgs({}).resolves(defaultUser);
29
```

```
30        const usersController = new UsersController(User);
31
32        await usersController.get(defaultRequest, response);
33        sinon.assert.calledWith(response.send, defaultUser);
34      });
35
36      it('should return 400 when an error occurs', async () => {
37        const request = {};
38        const response = {
39          send: sinon.spy(),
40          status: sinon.stub()
41        };
42
43        response.status.withArgs(400).returns(response);
44        User.find = sinon.stub();
45        User.find.withArgs({}).rejects({ message: 'Error' });
46
47        const usersController = new UsersController(User);
48
49        await usersController.get(request, response);
50        sinon.assert.calledWith(response.send, 'Error');
51      });
52    });
53
54    describe('getById()', () => {
55      it('should call send with one user', async () => {
56        const fakeId = 'a-fake-id';
57        const request = {
58          params: {
59            id: fakeId
60          }
61        };
62        const response = {
63          send: sinon.spy()
64        };
65
66        User.find = sinon.stub();
67        User.find.withArgs({ _id: fakeId }).resolves(defaultUser);
68
69        const usersController = new UsersController(User);
70
71        await usersController.getById(request, response);
72        sinon.assert.calledWith(response.send, defaultUser);
```

```
73      });
74    });
75
76    describe('create() user', () => {
77      it('should call send with a new user', async () => {
78        const requestWithBody = Object.assign(
79          {},
80          { body: defaultUser[0] },
81          defaultRequest
82        );
83        const response = {
84          send: sinon.spy(),
85          status: sinon.stub()
86        };
87        class fakeUser {
88          save() {}
89        }
90
91        response.status.withArgs(201).returns(response);
92        sinon
93          .stub(fakeUser.prototype, 'save')
94          .withArgs()
95          .resolves();
96
97        const usersController = new UsersController(fakeUser);
98
99        await usersController.create(requestWithBody, response);
100       sinon.assert.calledWith(response.send);
101     });
102
103     context('when an error occurs', () => {
104       it('should return 422', async () => {
105         const response = {
106           send: sinon.spy(),
107           status: sinon.stub()
108         };
109
110         class fakeUser {
111           save() {}
112         }
113
114         response.status.withArgs(422).returns(response);
115         sinon
```

```
116           .stub(fakeUser.prototype, 'save')
117           .withArgs()
118           .rejects({ message: 'Error' });
119
120         const usersController = new UsersController(fakeUser);
121
122         await usersController.create(defaultRequest, response);
123         sinon.assert.calledWith(response.status, 422);
124       });
125     });
126   });
127
128   describe('update() user', () => {
129     it('should respond with 200 when the user has been updated', async () => {
130       const fakeId = 'a-fake-id';
131       const updatedUser = {
132         _id: fakeId,
133         name: 'Updated User',
134         email: 'user@mail.com',
135         password: 'password',
136         role: 'user'
137       };
138       const request = {
139         params: {
140           id: fakeId
141         },
142         body: updatedUser
143       };
144       const response = {
145         sendStatus: sinon.spy()
146       };
147       class fakeUser {
148         static findById() {}
149         save() {}
150       }
151       const fakeUserInstance = new fakeUser();
152
153       const saveSpy = sinon.spy(fakeUser.prototype, 'save');
154       const findByIdStub = sinon.stub(fakeUser, 'findById');
155       findByIdStub.withArgs(fakeId).resolves(fakeUserInstance);
156
157       const usersController = new UsersController(fakeUser);
158
```

```
159       await usersController.update(request, response);
160       sinon.assert.calledWith(response.sendStatus, 200);
161       sinon.assert.calledOnce(saveSpy);
162     });
163
164     context('when an error occurs', () => {
165       it('should return 422', async () => {
166         const fakeId = 'a-fake-id';
167         const updatedUser = {
168           _id: fakeId,
169           name: 'Updated User',
170           email: 'user@mail.com',
171           password: 'password',
172           role: 'user'
173         };
174         const request = {
175           params: {
176             id: fakeId
177           },
178           body: updatedUser
179         };
180         const response = {
181           send: sinon.spy(),
182           status: sinon.stub()
183         };
184
185         class fakeUser {
186           static findById() {}
187         }
188
189         const findByIdStub = sinon.stub(fakeUser, 'findById');
190         findByIdStub.withArgs(fakeId).rejects({ message: 'Error' });
191         response.status.withArgs(422).returns(response);
192
193         const usersController = new UsersController(fakeUser);
194
195         await usersController.update(request, response);
196         sinon.assert.calledWith(response.send, 'Error');
197       });
198     });
199   });
200
201   describe('delete() user', () => {
```

```
202     it('should respond with 204 when the user has been deleted', async () => {
203       const fakeId = 'a-fake-id';
204       const request = {
205         params: {
206           id: fakeId
207         }
208       };
209       const response = {
210         sendStatus: sinon.spy()
211       };
212
213       class fakeUser {
214         static remove() {}
215       }
216
217       const removeStub = sinon.stub(fakeUser, 'remove');
218
219       removeStub.withArgs({ _id: fakeId }).resolves([1]);
220
221       const usersController = new UsersController(fakeUser);
222
223       await usersController.remove(request, response);
224       sinon.assert.calledWith(response.sendStatus, 204);
225     });
226
227     context('when an error occurs', () => {
228       it('should return 400', async () => {
229         const fakeId = 'a-fake-id';
230         const request = {
231           params: {
232             id: fakeId
233           }
234         };
235         const response = {
236           send: sinon.spy(),
237           status: sinon.stub()
238         };
239
240         class fakeUser {
241           static remove() {}
242         }
243
244         const removeStub = sinon.stub(fakeUser, 'remove');
```

```
245
246         removeStub.withArgs({ _id: fakeId }).rejects({ message: 'Error' });
247         response.status.withArgs(400).returns(response);
248
249         const usersController = new UsersController(fakeUser);
250
251         await usersController.remove(request, response);
252         sinon.assert.calledWith(response.send, 'Error');
253       });
254     });
255   });
256 });
```

O próximo passo é criar o controller para usuários em *src/controllers/users.js*, o arquivo deve ter o seguinte conteúdo:

```
1  class UsersController {
2    constructor(User) {
3      this.User = User;
4    }
5
6    async get(req, res) {
7      try {
8        const users = await this.User.find({});
9        res.send(users);
10     } catch (err) {
11       res.status(400).send(err.message);
12     }
13   }
14
15   async getById(req, res) {
16     const {
17       params: { id }
18     } = req;
19
20     try {
21       const user = await this.User.find({ _id: id });
22       res.send(user);
23     } catch (err) {
24       res.status(400).send(err.message);
25     }
26   }
27
```

```
28    async create(req, res) {
29      const user = new this.User(req.body);
30
31      try {
32        await user.save();
33        res.status(201).send(user);
34      } catch (err) {
35        res.status(422).send(err.message);
36      }
37    }
38
39    async update(req, res) {
40      const body = req.body;
41      try {
42        const user = await this.User.findById(req.params.id);
43
44        user.name = body.name;
45        user.email = body.email;
46        user.role = body.role;
47        if (body.password) {
48          user.password = body.password;
49        }
50        await user.save();
51
52        res.sendStatus(200);
53      } catch (err) {
54        res.status(422).send(err.message);
55      }
56    }
57
58    async remove(req, res) {
59      try {
60        await this.User.deleteOne({ _id: req.params.id });
61        res.sendStatus(204);
62      } catch (err) {
63        res.status(400).send(err.message);
64      }
65    }
66  }
67
68  export default UsersController;
```

No método update do UsersController temos um cenário diferente do controller de products, vamos

entender melhor o porquê em seguida. No momento, basta entender que vamos buscar o usuário do banco de dados e atualizar as propriedades. Caso o campo password esteja setado, ele também será atualizado. Este método update é referente ao método PUT do http. No PUT é esperado que seja enviado todos os campos que aparecem na requisição quando se faz um GET, por exemplo, se fizermos um get users/id a resposta vai conter `name`, `email`, `role` mas não deve conter o `password` por motivos de segurança. Sendo assim, o `password` só será recebido no `update` quando a intenção for alterar a senha, pois o campo não é obrigatório.

Depois de criar o controller é a hora de criarmos o Model em *src/models/user.js* com o seguinte trecho de código:

```
import mongoose from 'mongoose';

const schema = new mongoose.Schema({
  name: String,
  email: String,
  password: String,
  role: String
});

schema.set('toJSON', {
  transform: (doc, ret, options) => ({
    _id: ret._id,
    email: ret.email,
    name: ret.name,
    role: ret.role
  })
});

const User = mongoose.model('User', schema);

export default User;
```

No Model, além de criar o schema também sobrescrevemos o método `toJSON` que é responsável por transformar os dados que vem do MongoDB para o formato json; vamos utilizar a função transform, que é nativa do Mongoose, para remover o campo `password` do objeto final, pois não devemos expor a senha do usuário mesmo como hash. Sempre que o Mongoose faz uma busca no Mongo os dados vem em BSON o formato nativo do MongoDB, similar ao JSON só que binário. Depois de receber os dados o Mongoose faz o processo de serialização onde transforma o BSON que veio do banco em JSON para ser utilizado na aplicação, nesse momento é possível intervir nessa serialização e customizar o resultado final, exatamente o que implementamos no `toJSON`.

Com o Model pronto, vamos agora criar a rota em *src/routes/users.js*:

```
import express from 'express';
import UsersController from '../controllers/users';
import User from '../models/user';

const router = express.Router();
const usersController = new UsersController(User);
router.get('/', (req, res) => usersController.get(req, res));
router.get('/:id', (req, res) => usersController.getById(req, res));
router.post('/', (req, res) => usersController.create(req, res));
router.put('/:id', (req, res) => usersController.update(req, res));
router.delete('/:id', (req, res) => usersController.remove(req, res));

export default router;
```

Agora precisamos atualizar o index das rotas em *src/routes/index.js*, para carregar a rota de usuários:

```
import express from 'express';
import productsRoute from './products';
+import usersRoute from './users';

 const router = express.Router();

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

 export default router;
```

A próxima etapa será adicionar o arquivo de testes, mas, antes disso, vamos atualizar uma configuração nos nossos testes para que seja possível reutilizar a configuração do Supertest.

Vamos começar tornando global a rotina `before` que atualmente está disponível apenas para products. Vamos remover o trecho de código referente ao `before` do arquivo *test/integration/routes/products_spec.js*:

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

 describe('Routes: Products', () => {
-  let request;
-  let app;
-
-  before(async () => {
-    app = await setupApp();
-    request = supertest(app);
-  });
-
- after(async () => await app.database.connection.close());
-
   const defaultId = '56cb91bdc3464f14678934ca';
   const defaultProduct = {
     name: 'Default product',
```

E vamos inserir esse mesmo bloco nas definições globais, em um novo arquivo que vamos chamar de *test/integration/global.js*:

```
before(async () => {
  const app = await setupApp();
  global.app = app;
  global.request = supertest(app);
});

after(async () => await app.database.connection.close());
```

A última etapa da nossa refatoração é atualizar o arquivo *test/integration/mocha.opts* adicionando a chamada para o arquivo *global.js* que acabamos de criar:

```
--require @babel/register
- --require test/integration/helpers.js
+ --require test/integration/helpers.js test/integration/global.js
--reporter spec
--slow 5000
+ --timeout 5000
```

Dessa maneira o Mocha vai carregar esse arquivo global e executar o método before sempre antes de qualquer outro callback. Assim, o Supertest vai ser inicializado antes de todos os testes de integração. Também aumentamos o timeout para 5000ms, para evitar que algum teste de integração um pouco mais lento possa quebrar o nosso teste.

Agora basta criar os testes de integração para o modulo de users em *test/integration/routes/users_-spec.js*, o código será o seguinte:

```
import User from '../../../src/models/user';

describe('Routes: Users', () => {
  const defaultId = '56cb91bdc3464f14678934ca';
  const defaultAdmin = {
    name: 'Jhon Doe',
    email: 'jhon@mail.com',
    password: '123password',
    role: 'admin'
  };
  const expectedAdminUser = {
    _id: defaultId,
    name: 'Jhon Doe',
    email: 'jhon@mail.com',
    role: 'admin'
  };

  beforeEach(async() => {
    const user = new User(defaultAdmin);
    user._id = '56cb91bdc3464f14678934ca';
    await User.deleteMany({});
    await user.save();
  });

  afterEach(async() => await User.deleteMany({}));

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

      request
        .get('/users')
        .end((err, res) => {
          expect(res.body).to.eql([expectedAdminUser]);
          done(err);
        });
    });

    context('when an id is specified', done => {
      it('should return 200 with one user', done => {

        request
          .get(`/users/${defaultId}`)
          .end((err, res) => {
```

```
44              expect(res.statusCode).to.eql(200);
45              expect(res.body).to.eql([expectedAdminUser]);
46              done(err);
47            });
48        });
49      });
50    });
51
52    describe('POST /users', () => {
53      context('when posting an user', () => {
54
55        it('should return a new user with status code 201', done => {
56          const customId = '56cb91bdc3464f14678934ba';
57          const newUser = Object.assign({},{ _id: customId, __v:0 }, defaultAdmin);
58          const expectedSavedUser = {
59            _id: customId,
60            name: 'Jhon Doe',
61            email: 'jhon@mail.com',
62            role: 'admin'
63          };
64
65          request
66            .post('/users')
67            .send(newUser)
68            .end((err, res) => {
69              expect(res.statusCode).to.eql(201);
70              expect(res.body).to.eql(expectedSavedUser);
71              done(err);
72            });
73        });
74      });
75    });
76
77    describe('PUT /users/:id', () => {
78      context('when editing an user', () => {
79        it('should update the user and return 200 as status code', done => {
80          const customUser = {
81            name: 'Din Doe',
82          };
83          const updatedUser = Object.assign({}, defaultAdmin, customUser)
84
85          request
86            .put(`/users/${defaultId}`)
```

```
87            .send(updatedUser)
88            .end((err, res) => {
89              expect(res.status).to.eql(200);
90              done(err);
91            });
92        });
93      });
94    });
95
96    describe('DELETE /users/:id', () => {
97      context('when deleting an user', () => {
98        it('should delete an user and return 204 as status code', done => {
99
100         request
101           .delete(`/users/${defaultId}`)
102           .end((err, res) => {
103             expect(res.status).to.eql(204);
104             done(err);
105           });
106       });
107     });
108   });
109
110 });
```

Executando os testes:

```
1 $ npm test
```

Todos os testes devem estar passando, inclusive os testes de usuários.

Encriptando senhas com Bcrypt

Antes de começarmos a trabalhar na autenticação efetivamente, é necessário fazer mais uma melhoria na API. Note que adicionamos usuários, que possuem senhas, e estamos salvando as senhas diretamente no banco como texto.

Senhas em texto plano

Salvar senhas como texto é a maneira mais simples guardar essa informação, e também a mais insegura, pois, se um hacker tiver acesso ao servidor terá acesso às senhas dos usuários. Como as pessoas costumam utilizar a mesma senha para diferentes fins essa falha na nossa aplicação pode comprometer a segurança dos usuários.

Senhas com hashing de mão única

Hashing de mão única é uma prática de encriptação onde se encripta uma mensagem utilizando um algoritmo que não permite a desencriptação, bem mais seguro do que o texto plano. Porém, se o hacker descobrir o algoritmo utilizado pode fazer uso do mesmo para gerar senhas infinitamente, mais cedo ou mais tarde ele vai encontrar a certa, esse ataque é chamado de brute-force.

Exemplo: `SHA256("minhasenha") = "79809644A830EF92424A66227252B87BBDFB633A9DAB18BA450C1B8D35665F20"`

Senhas com hashing e salt

Hashing oferece mais segurança do que o texto plano, mas ataques como vimos acima podem acontecer. Uma solução para esse problema é utilizar um salt. Um salt nada mais é do que uma string concatenada a uma mensagem (a senha no nosso caso). Dessa maneira, havendo uma string única para a aplicação é possível gerar um hash que é muito difícil de ser quebrado por brute-force.

Exemplo: `SHA256("minhasenha" + "meusalt") = "697FDEADE02B2F4C86A5696D1DF998ADA97A6B1420F5BA0C7B4EE202`

Note que o hash gerado é diferente do exemplo anterior, para alguém gerar um hash igual a esse utilizando brute-force será necessário saber o salt. Ainda temos uma falha de segurança nesse cenário: Se alguém hackear o servidor e descobrir o salt conseguirá gerar hashs com brute-force que serão iguais aos gerados pela aplicação. Pode parecer muito difícil hackear o servidor, descobrir o salt, descobrir o algoritmo e quebrá-lo com brute-force, mas não é. Se você tem um produto aberto deve se preocupar muito com isso, hackers tentam esse tipo de coisa 24h por dia.

Criando senhas seguras com Bcrypt

Bcrypt é um algoritmo de hashing baseado em Blowfish[85] e com algumas características únicas, como a "key factor" que se refere a habilidade de aumentar a quantidade necessária de processamento para criptografar a informação. Aumentar a complexidade de processamento impossibilita a quebra de hashing por ataques como o brute-force por exemplo, pois o tempo necessário para gerar um hash similar é muito grande. O Bcrypt utiliza ainda um salt que é concatenado com o texto (nesse caso, a senha) para aumentar ainda mais a segurança e aleatoriedade do hash final gerado. Uma boa dica é utilizar um salt aleatório para cada senha gerada, isso garante que, mesmo que existam senhas iguais, elas não terão o mesmo hash final. Mas aí vem a pergunta: se vamos gerar hash aleatórios, como é possível verificar a senha do usuário no momento de fazer login? Mágica! Se passarmos a senha em texto plano (que o usuário vai fornecer na hora do login) e o hash gerado quando o usuário foi salvo no banco, o hash gerado pelo algoritmo será igual ao salvo no banco de dados. Vamos ver como isso funciona na prática, começando com a instalação do Bcrypt:

[85] https://en.wikipedia.org/wiki/Blowfish_(cipher)

```
$ npm install bcrypt@^3.0.7
```

Após a instalação do Bcrypt vamos atualizar o Model de user adicionando o seguinte:

```
import mongoose from 'mongoose';
+import Util from 'util';
+import bcrypt from 'bcrypt';

+const hashAsync = Util.promisify(bcrypt.hash);
const schema = new mongoose.Schema({
  name: String,
  email: String,
  password: String,
  role: String
});

+schema.pre('save', async function(next) {
+  if (!this.password || !this.isModified('password')) {
+    return next();
+  }
+  try {
+    const hashedPassword = await hashAsync(this.password, 10);
+    this.password = hashedPassword;
+  } catch (error) {
+    next(err);
+  }
+});
```

Muita coisa acontece nesse bloco de código, vamos começar pelo Bcrypt. O módulo nativo do Bcrypt não suporta promises, ou seja, teríamos que usar callbacks, mas como em todo nosso código utilizamos promises vamos seguir o padrão. Atualmente é simples transformar uma função que utiliza callback para se comportar como uma Promise, basta utilizar o módulo nativo util do Node.js e chamar o método promisify[86] passando a referência da função que utiliza callback, como fizemos com `bcrypt.hash`, o retorno será uma função que utiliza Promise.

Middlewares no Mongoose

Para garantir que sempre que um usuário for salvo a senha dele será encriptada vamos utilizar uma funcionalidade do Mongoose chamada middlewares[87] (também conhecido como pre e post

[86] https://nodejs.org/dist/latest-v8.x/docs/api/util.html#util_util_promisify_original
[87] http://mongoosejs.com/docs/middleware.html

hooks). No trecho de código anterior utilizamos o pre save, ou seja, esse código será automaticamente executado sempre antes da função save do Model. Começamos verificando se o campo password foi realmente alterado:

```
  if(!this.password || !this.isModified('password')) {
    return next();
  };
```

Se o campo password não foi alterado não podemos gerar um hash novo, se não estaríamos gerando um hash de um hash e o usuário não conseguiria mais utilizar a senha. Caso o campo password tenha sido alterado, o trecho de código acima vai gerar um hash para a nova senha do usuário e substituir a antiga no Model, na sequência o Model vai salvar o hash ao invés da senha em texto plano que o usuário enviou. A função hashAsync é a função do Bcrypt que transformamos em Promise, ela será responsável por criar um hash a partir da senha que o usuário enviou.

Além da senha em texto, também passamos número 10 para o Brcrypt, esse número se refere ao factor; o factor é utilizado para dizer ao Bcrypt o número de complexidade que desejamos para gerar o hash, quanto maior for o número mais tempo ele vai levar para gerar o hash e mais difícil será para desencriptar. Na sequência substituimos o password que o usuário enviou pelo hash:

```
      this.password = hashedPassword;
```

O this nesse contexto se refere ao Model do Mongoose, como estamos utilizando um middleware no momento que chamamos o next ele vai chamar a próxima ação da cadeia de middlewares, que provavelmente será a ação de save, caso não exista outro middleware, então o usuário será salvo no banco com o password como hash.

Agora que temos toda a lógica necessária para criar senhas com segurança vamos voltar ao método `update` do UsersController para entendê-lo melhor:

```
  async update(req, res) {
    const body = req.body;
    try {
      const user = await this.User.findById(req.params.id);

      user.name = body.name;
      user.email = body.email;
      user.role = body.role;
      if (body.password) {
        user.password = body.password;
      }
      await user.save();

```

```
      res.sendStatus(200);
    } catch (err) {
      res.status(422).send(err.message);
    }
  }
```

Não vamos alterar nada no código, apenas destaquei o método update novamente para explicar o seu comportamento. Provavelmente alguns de vocês leitores já trabalharam com MongoDB e sabem que ele possui um método de update e nesse caso estamos buscando o usuário do banco, atualizando os dados e chamando o método save. Isso acontece por que o update nativo do MongoDB é baixo nível, ou seja, ele não pertence ao Mongoose, nós não podemos utilizar o middleware `schema.pre('update')` por exemplo para ter acesso aos dados anteriores e aos novos que estão sendo salvos.

Esse comportamento tem uma razão, o MongoDB como explicado anteriormente, é um banco de dados NoSQL, ou seja, ele não garante integridade dos dados. Vamos analisar um exemplo do método update do Mongoose:

```
  async update(req, res) {
    try {
      await this.Product.updateOne({ _id: req.params.id }, req.body);
      res.sendStatus(200);
    } catch (err) {
      res.status(422).send(err.message);
    }
  }
```

Esse é o ProductsController, como não utilizamos nenhum middleware do Mongoose nesse `update` podemos utilizar o método `update` nativo do MongoDB. O que acontece internamente no `findOneAndUpdate` é o seguinte:

```
  db.products.update(
  { _id: "example-id" },
    {
      name: "Updated Name",
    }
)
```

O Mongoose não busca o produto para atualizar e salvar novamente, ele apenas traduz o `updateOne` em uma query nativa de update do MongoDB, sendo assim mesmo que seja adicionado um middleware no pre update não teremos acesso aos dados anteriores para comparar se a senha do usuário mudou ou não e não saberemos se é necessário gerar outro hash.

Antes de pensar que esse comportamento do Mongoose é “burro” vamos lembrar das premissas do MongoDB enquanto NoSQL é não prover Atomicidade e Consistência, isso significa que se o Moongose buscar o produto para a memória, atualizar os campos que mudaram e salvar novamente ele teria que lidar com concorrência, pois, imagine que nesse meio tempo em que o produto está em memória sendo atualizado outro processo também tentar atualizar o mesmo produto, isso causaria inconsistência. Por isso o Mongoose delega essa responsabilidade para o MongoDB que nativamente trata os updates de forma sequencial.

O código desse capitulo está disponvivel aqui[88].

[88] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step10

Autenticação e controle de acesso com Access Control List - ACL

ACL ou Access Control List, é o termo utilizado na engenharia de software para falar de Controle de Acesso a Sistemas. A maioria das implementações de ACL seguem um padrão estruturado, onde se define as ações que usuários podem executar no sistema, como por exemplo:

```
{
        John: read,
        Meg: admin
}
```

Assim o sistema checa as permissões do usuário e decide se permite ou não que ele execute determinada ação. No Exemplo acima o usuário John tem permissão de leitura e Meg tem permissão de administrador.

O padrão de ACL pode ser aplicado a quase todo tipo de sistema onde é necessário controlar ações de usuários, incluindo APIs.

Como nossa API utiliza express, vamos usar o módulo express-acl para implementar o ACL em nossa aplicação.

Express ACL

O express-acl[89] é um módulo feito para o express que realiza checagem de acesso em runtime e pode ser configurado via JSON ou YAML (aqui vamos optar por JSON). Vamos começar instalando o módulo:

```
$ npm install express-acl@2.0.2
```

Vamos criar um arquivo chamado *nacl.json* no diretório de configuração *config/nacl.json* e adicione o seguinte trecho de código:

[89] https://nyambati.github.io/express-acl/

```
[
    {
        "group": "admin",
        "permissions": [
            {
                "resource": "*",
                "methods": "*",
                "action": "allow"
            }
        ]
    },
    {
        "group": "user",
        "permissions": [
            {
                "resource": "products",
                "methods": [
                    "GET"
                ],
                "action": "allow"
            }
        ]
    }
]
```

Ainda não temos todas as configurações necessárias mas já vamos deixar pronta a parte do acl. O express acl funciona por grupos: permitindo ou não ações para um determinado grupo a um determinado recurso da API. Por exemplo, o primeiro grupo será o de administrador: admin, os administradores terão acesso a todos os recursos da API, sem limitação alguma, para isso é utilizado * com a `"action":"allow"`, isso significa que a ação será de "permitir" em todos os recursos (resource) e metodos (methods).

O segundo grupo é o de usuário (user), usuários somente terão acesso para leitura dos produtos da API.Como já vimos,a leitura é feita pelo verbo http GET, então para usuários precisamos permitir acesso ao método GET do recurso products. Como feito neste bloco:

```
        "group": "user",
        "permissions": [
            {
                "resource": "products",
                "methods": [
                    "GET"
                ],
                "action": "allow"
            }
```

Agora vamos atualizar o arquivo *src/app.js* para adicionar a chamada ao acl.

```
+ import acl from 'express-acl';

const app = express();

+acl.config({
+  baseUrl: '/',
+  path: 'config'
+});

const configureExpress = () => {
  app.use(bodyParser.json());
+  app.use(acl.authorize.unless({path:['/users/authenticate']}));

  app.use('/', routes);
  app.database = database;

  return app;
};
```

Note que no `app.use(acl.authorize.unless({path:['/users/authenticate']}));` estamos passando o express-acl como um middleware para o express, além disso adicionamos a configuração `authorize.unless` que diz para o express acl validar todos os recursos da API unless (a não ser que) seja `/users/authenticate`, essa rota será pública e utilizada para gerar o token de autenticação.

Autenticação com JSON Web Token

Primeiro falamos de autorização com express ACL e agora vamos trabalhar na autenticação. A diferença entre autenticação e autorização é que na autenticação verificamos a identidade de um usuário e na autorização verificamos se o usuário autenticado tem privilégios para executar

determinada ação. Autenticação pode ser feita de várias maneiras, a mais utilizada é o login com usuário e senha que cria uma sessão. Como APIs devem ser stateless, ou seja, não devem armazenar estado, não é possível ter sessão, dado que para isso ela teria que ser armazenada no servidor.

Quando nada é armazenado no servidor fica mais fácil escalar a aplicação pois ela não tem estado em lugar algum, o usuário que controla o estado. Para resolver esse problema foram criados os tokens (também é comum utilizar cookies para autenticação). Utilizando tokens o usuário que faz a requisição autêntica uma vez com as credenciais, (no nosso caso email e senha) e recebe um token que será usado para fazer requisições para a API.

Existem várias maneiras de fazer autenticação baseada em token, como JSON Web Token e OAuth. Aqui utilizamos JSON Web Token.

JSON Web Token[90], JWT[91], é um padrão aberto RFC 7519 que define uma maneira compacta de transportar objetos JSON seguramente entre partes. A confiança e segurança é alcançada por meio de assinatura digital utilizando algoritmos como HMAC ou chave pública/privada RSA ou ECDSA. Utilizaremos um modulo npm chamado jsonwebtoken que implementa a spec oficial do JWT e nos permite gerar e validar tokens no Node.js. Vamos começar pela instalação do módulo:

```
1 $ npm install jsonwebtoken@8.3.0
```

Adicionaremos duas propriedades necessárias para a configuração em *config/default.json*, primeiro a propriedade key dentro do objeto auth, essa será a chave secreta utilizada para assinar o token, e a propriedade tokenExpiresIn que se refere ao tempo de expiração do token. Na configuração abaixo definimos 7 dias, após esse prazo o usuário precisa gerar um novo token.

```
1 {
2   "database": {
3     "mongoUrl": "mongodb://localhost:27017/shop"
4   },
5 +   "auth": {
6 +    "key": "thisisaverysecurekey",
7 +    "tokenExpiresIn": "7d"
8 +  }
9 }
```

Criando Middlewares

Como será necessário validar o token do usuário em todas requisições vamos criar um middleware responsável por validar se a requisição possui um token, se sim, vamos decodificar o token e transformá-lo em um objeto que será adicionado na requisição para ser utilizado posteriormente pelo express-acl. Antes de tudo vamos começar pelo teste de unidade do middleware criando um arquivo em *test/unit/middlewares/auth_spec.js*.

[90] https://jwt.io/
[91] https://tools.ietf.org/html/rfc7519

```
import authMiddleware from '../../../src/middlewares/auth';
import jwt from 'jsonwebtoken';
import config from 'config';

describe('AuthMiddleware', () => {
  it('should verify a JWT token and call the next middleware', done => {
    const jwtToken = jwt.sign({ data: 'fake' }, config.get('auth.key'));
    const reqFake = {
      headers: {
        'x-access-token': jwtToken
      }
    };
    const resFake = {};
    authMiddleware(reqFake, resFake, done);
  });
});
```

Aqui começamos pelo primeiro happy path, o middleware deve receber uma requisição, verificar o token e chamar o próximo middleware. Note que aqui `jwt.sign({ data: 'fake' }, config.get('auth.key'));` geramos um token fake para ser utilizado no teste, esse token segue a mesma lógica que a aplicação utiliza. Para validar o teste chamamos `authMiddleware(reqFake, resFake, done);` passamos o reqFake, que simula uma requisição contendo o header com o JWT, e um resFake vazio simulando o objeto de response que o middleware espera. Passamos também o callback done do Mocha como o next do middleware, dessa maneira quando o authMiddleware chamar o próximo middleware ele vai estar chamando o done do Mocha finalizando o teste.

Executando os testes de unidade agora teremos um erro

```
$ npm run test:unit

Error: Cannot find module '../../../src/middlewares/auth'
    at Function.Module._resolveFilename (internal/modules/cjs/loader.js:548:15)
    at Function.Module._load (internal/modules/cjs/loader.js:475:25)
```

O arquivo não foi encontrado, crie o arquivo em */src/middlewares/auth.js* com o seguinte código:

```
export default (req, res, next) => {
  next()
};
```

Aqui adicionamos lógica somente para o teste passar, passo green do TDD (espero que ainda lembrem!).

Executando os testes novamente a saída será:

```
  AuthMiddleware
    ✓ should verify a JWT token and call the next middleware
```

Agora é hora de aplicar a lógica de verdade. Altere o arquivo *auth.js* como abaixo:

```
+import jwt from 'jsonwebtoken';
+import config from 'config';

export default (req, res, next) => {
+  const token = req.headers['x-access-token'];
+
+  jwt.verify(token, config.get('auth.key'), (err, decoded) => {
+    req.decoded = decoded;
+    next(err);
+  });
};
```

Primeiro passo pegamos o token x-access-token do header da requisição que e depois o verificamos utilizando o módulo jsonwebtoken. O primeiro parâmetro é o token, o segundo é a chave secreta da nossa aplicação para poder decodificar o token e o terceiro parâmetro é o callback que o jsonwebtoken espera. Em seguida adicionamos o token decodificado ao objeto req referente a requisição `req.decoded = decoded` e chamamos o próximo middleware com next. Note que o err é passado como parâmetro para o próximo middleware, isso significa que se ocorrer algum erro na hora de decodificar o token o jsonwebtoken vai passar esse erro para nós e nós vamos passá-lo adiante para o próximo middleware, no futuro teremos um middleware somente para tratar erros.

Executando os testes de unidade novamente:

```
$ npm run test:unit

  AuthMiddleware
    ✓ should verify a JWT token and call the next middleware
```

É um pouco confuso passar o *err* para o próximo middleware, certo? Isso significa que nosso código possui dois caminhos, um de sucesso e um de falha, então devemos testar ambos. Vamos escrever um teste que simula um caso de erro adicionando o seguinte caso de teste em *test/unit/middlewares/auth_spec.js*.

```
import authMiddleware from '../../../src/middlewares/auth';
import jwt from 'jsonwebtoken';
import config from 'config';

describe('AuthMiddleware', () => {

    it('should verify a JWT token and call the next middleware', done => {
        const jwtToken = jwt.sign({ data: 'fake' }, config.get('auth.key'));
        const reqFake = {
            headers: {
                'x-access-token': jwtToken
            }
        };
        const resFake = {};
        authMiddleware(reqFake, resFake, done);
    });

+    it('should call the next middleware passing an error when the token validation \
fails', done => {
+        const reqFake = {
+           headers: {
+                'x-access-token': 'invalid token'
+            }
+        };
+        const resFake = {};
+       authMiddleware(reqFake, resFake, err => {
+            expect(err.message).to.eq('jwt malformed');
+            done();
+        });
+    });
});
```

Passando um valor qualquer no header x-access-token fará com que o jsonwebtoken falhe e o nosso código var chamar o next passando o erro recebido pelo jsonwebtoken. No teste basta checarmos a mensagem: `expect(err.message).to.eq('jwt malformed')` "jwt malformed' é a mensagem lançada pelo jsonwebtoken quando ele recebe um token que não segue o padrão do JWT.

Executando os testes, a saída será:

```
  AuthMiddleware
    ✓ should verify a JWT token and call the next middleware
    ✓ should call the next middleware passing an error when the token validation fai\
ls
```

Ainda temos um caso para testar: na implementação atual o código espera que toda a requisição envie um token, mas a requisição para gerar o token não tem como passar um token!. Sendo assim, nosso código precisa verificar se existe token na requisição e chamar o próximo middleware sem executar a lógica de decodificação do jsonwebtoken. Vamos para o teste:

```
 it('should call next middleware if theres no token', done => {
        const reqFake = {
            headers: {}
        };
        const resFake = {};
        authMiddleware(reqFake, resFake, done);
    });
```

Adicione o teste acima no teste unitário do auth middleware. Nele não var ser passado o header x-access-token, executando o teste:

```
$npm run test:unit

  AuthMiddleware
    ✓ should verify a JWT token and call the next middleware
    ✓ should call the next middleware passing an error when the token validation fai\
ls
    1) should call next middleware if theres no token


  2 passing (16ms)
  1 failing

  1) AuthMiddleware should call next middleware if theres no token:
     JsonWebTokenError: jwt must be provided
      at Object.module.exports [as verify] (node_modules/jsonwebtoken/verify.js:39:1\
7)
      at exports.default (src/middlewares/auth.js:8:7)
      at Context.done (test/unit/middlewares/auth_spec.js:36:9)
```

O teste vai quebrar. A mensagem é jwt must be provided o que significa que o código tentou verificar o token, vamos alterar o código para não verificar o token quando ele não estiver na requisição. Altere o middleware *auth.js*:

```
import jwt from 'jsonwebtoken';
import config from 'config';

export default (req, res, next) => {
  const token = req.headers['x-access-token'];
+  if (!token) {
+    return next();
+  };
  jwt.verify(token, config.get('auth.key'), (err, decoded) => {
    req.decoded = decoded;
    next(err);
  });
};
```

Simples! Se não tiver token chamamos o próximo middleware. Os testes agora devem estar passando

```
  AuthMiddleware
    ✓ should verify a JWT token and call the next middleware
    ✓ should call the next middleware passing an error when the token validation fai\
ls
    ✓ should call next middleware if theres no token
```

Nosso middleware esta pronto.

O próximo passo é alterar o *app.js* adicionando o auth middleware ao express.

```
import express from 'express';
import bodyParser from 'body-parser';
import acl from 'express-acl';
import routes from './routes';
import database from '../config/database';
+import authMiddleware from './middlewares/auth.js';

 const app = express();

acl.config({
  baseUrl:'/',
  path: 'config'
});

const configureExpress = () => {
  app.use(bodyParser.json());
+  app.use(authMiddleware);
```

```
18   app.use(acl.authorize.unless({path:['/users/authenticate']}));
19
20   app.use('/', routes);
21   app.database = database;
22
23   return app;
24 };
```

Os testes de integração não estão passando pois ainda não implementamos a autenticação. Vamos começar a implementação da autenticação pelo teste da rota em *test/integration/users_spec.js*

```
1  +describe('POST /users/authenticate', () => {
2  +    context('when authenticating an user', () => {
3  +      it('should generate a valid token', done => {
4  +
5  +       request
6  +         .post(`/users/authenticate`)
7  +         .send({
8  +           email: 'jhon@mail.com',
9  +           password: '123password'
10 +         })
11 +         .end((err, res) => {
12 +           expect(res.body).to.have.key('token');
13 +           expect(res.status).to.eql(200);
14 +           done(err);
15 +         });
16 +     });
17
18 +     it('should return unauthorized when the password does not match', done => {
19
20 +       request
21 +         .post(`/users/authenticate`)
22 +         .send({
23 +           email: 'jhon@mail.com',
24 +           password: 'wrongpassword'
25 +         })
26 +         .end((err, res) => {
27 +           expect(res.status).to.eql(401);
28 +           done(err);
29 +         });
30 +     });
31 +   });
32 + });
```

Adicionamos 2 testes de integração, um para o caso de sucesso, onde espera-se que o usuário seja autenticado, e um para o caso de falha, onde não foi possível autenticar o usuário. Vamos executar os testes de integração:

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

Ignore os outros testes nesse momento, vamos focar apenas nos novos testes:

```
9) Routes: Users POST /users/authenticate when authenticating a user should generate\
 a valid token:

      Uncaught AssertionError: expected {} to have key 'token'
      + expected - actual

      -[]
      +[
      +  "token"
      +]

      at Test.request.post.send.end (test/integration/routes/users_spec.js:120:38)
      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:1656:8)
      at process._tickCallback (internal/process/next_tick.js:178:19)

  10) Routes: Users POST /users/authenticate when authenticating an user should retu\
rn unauthorized when the password does not match:

      Uncaught AssertionError: expected 404 to deeply equal 401
      + expected - actual

      -404
      +401

      at Test.request.post.send.end (test/integration/routes/users_spec.js:135:35)
      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:1656:8)
      at process._tickCallback (internal/process/next_tick.js:178:19)
```

Estamos recebendo 404 por que a rota não existe, vamos criá-la em *src/users/routes/users.js* adicionando o seguinte código:

```
import express from 'express';
import UsersController from '../controllers/users';
import User from '../models/user';

const router = express.Router();
const usersController = new UsersController(User);

router.get('/', (req, res) => usersController.get(req, res));
router.get('/:id', (req, res) => usersController.getById(req, res));
router.post('/', (req, res) => usersController.create(req, res));
router.put('/:id', (req, res) => usersController.update(req, res));
router.delete('/:id', (req, res) => usersController.remove(req, res));
+router.post('/authenticate', (req, res) => usersController.authenticate(req, res));

export default router;
```

Se executarmos os testes novamente a saída será:

```
9) Routes: Users POST /users/authenticate when authenticating an user should generat\
e a valid token:

      Uncaught AssertionError: expected {} to have key 'token'
      + expected - actual

      -[]
      +[
      +  "token"
      +]

      at Test.request.post.send.end (test/integration/routes/users_spec.js:120:38)
      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:1656:8)
      at process._tickCallback (internal/process/next_tick.js:178:19)

  10) Routes: Users POST /users/authenticate when authenticating a user should retur\
n unauthorized when the password does not match:

      Uncaught AssertionError: expected 500 to deeply equal 401
      + expected - actual

      -500
      +401
```

```
26
27        at Test.request.post.send.end (test/integration/routes/users_spec.js:135:35)
28        at Test.assert (node_modules/supertest/lib/test.js:179:6)
29        at Server.assert (node_modules/supertest/lib/test.js:131:12)
30        at emitCloseNT (net.js:1656:8)
31        at process._tickCallback (internal/process/next_tick.js:178:19)
```

Executando os testes de integração novamente a saída será um erro 500, pois não temos o método authenticate no controller de users, vamos criá-lo começando com um teste unitário em *test/unit/controllers/users_spec.js*. O método authenticate recebe os parâmetros req e res. Esperamos que método chame o objeto res do express passando o token como resposta. Abaixo vamos ver como testar esse cenário:

```
1   describe('authenticate', () => {
2        it('should authenticate a user', done => {
3       const fakeReq = {
4         body: {}
5       };
6       const fakeRes = {
7         send: token => {
8           expect(token).to.eql({ token: 'fake-token' });
9           done();
10        }
11      };
12      const usersController = new UsersController({});
13      usersController
14        .authenticate(fakeReq, fakeRes);
15    });
16  });
```

O teste é conciso pois neste momento não sabemos como nosso futuro código será, apenas sabemos que a saída deve ser um token. Aqui seguiremos o TDD de forma evolutiva para entender como o design do código muda durante o desenvolvimento. É importante atentar no código acima para o fakeRes ele possui um método send que imita o objeto real do express e dentro desse método adicionamos o expect do teste. Esse método é um callback então quando ele for chamado no final do fluxo o expect será executado e saberemos se o teste passou.

Agora vamos implementar o metodo authenticate em *src/controllers/users*

```
  async authenticate(req, res) {
    return res.send({ token: 'fake-token' });
  }
```

Aqui estamos no passo GREEN do TDD: o suficiente para o teste passar. Nosso código ainda não está gerando o token, vamos para o REFACTOR para aplicar a lógica necessária. Nesse caso eu já fiz alguns testes e imagino como a implementação vai ser, por isso posso escrever o teste:

É importante notar que em um cenário real vamos testar várias implementações antes de pensar em como escrever o teste. Quando falamos sobre TDD falamos sobre escrever o teste antes e isso causa confusão. Antes de escrever o teste é necessário experimentar com o código, uma vez que tivermos uma ideia de como será o código podemos começar o teste.

Primeiramente vamos adicionar os seguintes imports:

```
import UsersController from '../../../src/controllers/users';
import sinon from 'sinon';
+import jwt from 'jsonwebtoken';
+import config from 'config';
+import bcrypt from 'bcrypt';
import User from '../../../src/models/user';


    it('should authenticate a user', async() => {
      const fakeUserModel = {
        findOne: sinon.stub()
      };
      const user = {
        name: 'Jhon Doe',
        email: 'jhondoe@mail.com',
        password: '12345',
        role: 'admin'
      };
      const userWithEncryptedPassword = {...user, password: bcrypt.hashSync(user.pas\
sword, 10) };
      fakeUserModel.findOne.withArgs({ email: user.email }).resolves({
        ...userWithEncryptedPassword,
        toJSON: () => ({ email: user.email })
      });

      const jwtToken = jwt.sign(userWithEncryptedPassword, config.get('auth.key'), {
        expiresIn: config.get('auth.tokenExpiresIn')
```

```
28        });
29        const fakeReq = {
30          body: user
31        };
32        const fakeRes = {
33          send: sinon.spy()
34        };
35        const usersController = new UsersController(fakeUserModel);
36        await usersController.authenticate(fakeReq, fakeRes);
37        sinon.assert.calledWith(fakeRes.send, { token: jwtToken });
38      });
```

Quando enviados usuário e senha um json web token é gerado, este é o cenário que o teste acima representa. Se o teste não estiver fazendo sentido, não se preocupe, essa parte será mais fácil de entender quando chegarmos na implementação. Dois pontos são importantes no teste acima:

```
1 const userWithEncryptedPassword = {...user, password: bcrypt.hashSync(user.password,\
2  10) };
```

Aqui geramos um hash da senha para poder simular um usuário no banco de dados, para nao tem que escrever todas as propriedades do objeto user novamente utilizamos spread operator[92] para clonar o objecto utilizando ... na frente do objeto que queremos clonar e substituindo a propriedade password pela nova com o valor encriptado pelo bcrypt.

Na sequência, usando o stub no método findOne do fakeUserModel o usuário é retornado com o password. O método toJSON é usado para simular o método existente no Mongoose.

```
1         fakeUserModel.findOne.withArgs({ email: user.email }).resolves({
2           ...userWithEncryptedPassword,
3           toJSON: () => ({ email: user.email })
4         });
```

O próximo é:

```
1         const jwtToken = jwt.sign(userWithEncryptedPassword, config.get('auth.key'), {
2           expiresIn: config.get('auth.tokenExpiresIn')
3         });
```

Aqui é gerado um JWT baseado nos dados falsos de usuário e com a senha encriptada, assim devemos ter o mesmo JWT que o código vai gerar e será possível comparar o token no teste.

Agora vamos fazer a implementação do código no usersController, começando pela importação das bibliotecas:

[92] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

```
+ import jwt from 'jsonwebtoken';
+ import config from 'config';
+ import bcrypt from 'bcrypt';
```

Agora, vamos reescrever o método authenticate:

```
  async authenticate(req, res) {
    const { email, password } = req.body;
    const user = await this.User.findOne({ email });
    if(!user.password == bcrypt.compareSync(password, user.password)) {
      // To be implemented
      return;
    }
    const token = jwt.sign(
      {
        name: user.name,
        email: user.email,
        password: user.password,
        role: user.role
      },
      config.get('auth.key'),
      {
        expiresIn: config.get('auth.tokenExpiresIn')
      }
    );
    res.send({ token });
  }
```

A logica `!user.password == bcrypt.compareSync(password, user.password)` foi adicionada somente para prevenir o teste de passar caso as senhas nao sejam iguais, a implementação sera feita depois.

O código a seguir busca um usuário no banco de dados baseado no email provido, em seguida compara as senhas e caso sejam iguais será gerado um token para o usuário, caso contrário retornará um erro. Agora os testes de unidade devem estar passando.

```
$ npm run test:unit

  Controller: Users
    authenticate
      ✓ should authenticate a user
```

Agora vamos aos testes de integração:

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

A saida sera:

```
3 passing (689ms)
9 failing
```

Os testes devem estar falhando pois as rotas esperam receber um token. Vamos deixar os testes de integração de lado nesse momento e seguir nossa implementação interna. Os testes de integração servem para garantir que a integração entre os componentes esta funcionando, vamos voltar a eles quando terminarmos a implementação interna dos testes de unidade.

Voltando aos testes unitários, o próximo passo é testar um caminho de erro. Um caminho de erro pode ser quando o usuário não for encontrado ou quando a senha não bater. Vamos adicionar um novo caso de teste:

```
    it('should return 401 when the user can not be found', async () => {
      const fakeUserModel = {
        findOne: sinon.stub()
      };
      fakeUserModel.findOne.resolves(null);
      const user = {
        name: 'Jhon Doe',
        email: 'jhondoe@mail.com',
        password: '12345',
        role: 'admin'
      };
      const fakeReq = {
        body: user
      };
      const fakeRes = {
        sendStatus: sinon.spy()
      };
      const usersController = new UsersController(fakeUserModel);

```

```
      await usersController.authenticate(fakeReq, fakeRes);
      sinon.assert.calledWith(fakeRes.sendStatus, 401);
    });
```

Executando o teste teremos o seguinte erro:

```
 25 passing (264ms)
  1 failing

  1) Controller: should return 401 when the user can not be found:
     TypeError: Cannot read property 'toJSON' of null
      at User.findOne.then.user (src/controllers/users.js:62:37)
```

Vamos à implementação:

```
  async authenticate(req, res) {
    const { email, password } = req.body;
+    try {
      const user = await this.User.findOne({ email });
      if (!user.password == bcrypt.compareSync(password, user.password)) {
-      // To be implemented
-      return;
+        throw new Error('User Unauthorized');
      }
      const token = jwt.sign(
        {
          name: user.name,
          email: user.email,
          password: user.password,
          role: user.role
        },
        config.get('auth.key'),
        {
          expiresIn: config.get('auth.tokenExpiresIn')
        }
      );
      res.send({ token });
+    } catch (err) {
+      res.sendStatus(401);
+    }
  }
```

Não estamos passando pelo passo GREEN do TDD, estamos fazendo a implementação direta pois, neste caso, é bem simples. Aqui verificamos se o usuário existe, se não existir retornamos um erro 401.

Os testes devem estar passando agora, podemos seguir em frente e testar o comportamento da senha:

```
    it('should return 401 when the password does not match', async () => {
      const fakeUserModel = {
        findOne: sinon.stub()
      };
      const user = {
        name: 'Jhon Doe',
        email: 'jhondoe@mail.com',
        password: '12345',
        role: 'admin'
      };
      const userWithDifferentPassword = {
        ...user,
        password: bcrypt.hashSync('another_password', 10)
      };
      fakeUserModel.findOne.withArgs({ email: user.email }).resolves({
        ...userWithDifferentPassword
      });
      const fakeReq = {
        body: user
      };
      const fakeRes = {
        sendStatus: sinon.spy()
      };
      const usersController = new UsersController(fakeUserModel);

      await usersController.authenticate(fakeReq, fakeRes);
      sinon.assert.calledWith(fakeRes.sendStatus, 401);
    });
```

A única parte diferente nesse bloco de código é a seguinte:

```
      const userWithDifferentPassword = {
        ...user,
        password: bcrypt.hashSync('another_password', 10)
      };
      fakeUserModel.findOne.withArgs({ email: user.email }).resolves({
        ...userWithDifferentPassword
      });
```

Aqui simulamos um cenário onde as senhas não batem. Quando isso acontece devemos retornar um erro 401 para o usuário informando que ele não pôde ser autenticado.

Neste momento os testes devem estar passando.

O próximo passo será extrair a lógica de autenticação para um serviço separado, para isso vamos começar pelo teste unitário em *test/unit/services/auth_spec.js*:

```
import AuthService from '../../../src/services/auth';


describe('Service: Auth', () => {
  context('authenticate', () => {
    it('should authenticate an user', () => {
    });
  });
});
```

No primeiro passo criamos o arquivo do teste com apenas o mínimo e por isso teremos o seguinte erro:

```
Error: Cannot find module '../../../src/services/auth'
```

Vamos criar o AuthService em *src/services/auth.js*, e então vamos melhorar nosso teste adicionando o comportamento que esperamos:

```
import AuthService from '../../../src/services/auth';
import bcrypt from 'bcrypt';
import Util from 'util';
import sinon from 'sinon';

const hashAsync = Util.promisify(bcrypt.hash);

describe('Service: Auth', () => {
  context('authenticate', () => {
```

```
10      it('should authenticate a user', async() => {
11        const fakeUserModel = {
12          findOne: sinon.stub()
13        };
14        const user = {
15          name: 'John',
16          email: 'jhondoe@mail.com',
17          password: '12345'
18        };
19
20        const authService = new AuthService(fakeUserModel);
21        const hashedPassword = await hashAsync('12345', 10);
22        const userFromDatabase = { ...user,
23          password: hashedPassword
24        };
25
26        fakeUserModel.findOne.withArgs({ email: 'jhondoe@mail.com' }).resolves(userFro\
27 mDatabase);
28
29        const res = await authService.authenticate(user);
30
31        expect(res).to.eql(userFromDatabase);
32      });
33    });
34 });
```

Não há nada de novo nesse cenário, é o mesmo teste feito no users controller authenticate, então vamos copiar a lógica do users_controller no método authenticate. O código deve ficar assim no AuthService:

```
1  import bcrypt from 'bcrypt';
2  import jwt from 'jsonwebtoken';
3  import config from 'config';
4
5  class Auth {
6    constructor(User) {
7      this.User = User;
8    }
9
10   async authenticate(data) {
11     const user = await this.User.findOne({email: data.email});
12
13     if(!user || !(await bcrypt.compare(data.password, user.password))) {
```

```
      return false;
    }

    return user;
  }
}

export default Auth;
```

Os testes de unidade agora devem estar passando:

```
Service: Auth
    authenticate
      ✓ should authenticate an user
```

Note que mudamos o `bcrypt.compareSync` para `bcrypt.compare` que é assincrono, dessa maneira não bloqueamos o Event Loop. Note também que os testes não devem quebrar com essa alteração.

Vamos adicionar mais um teste de unidade para o caso onde as senhas não batem:

```
    it('should return false when the password does not match', async () => {
      const user = {
        email: 'jhondoe@mail.com',
        password: '12345'
      };
      const fakeUserModel = {
        findOne: sinon.stub()
      };
      fakeUserModel.findOne.resolves({ email: user.email, password: 'aFakeHashedPass\
word' });
      const authService = new AuthService(fakeUserModel);
      const response = await authService.authenticate(user);

      expect(response).to.be.false;
    });
```

Com os teste de unidade prontos o próximo passo será atualizar o código no método authenticate do users controller, para utilizar o AuthService. Para isso serão necessárias algumas alterações, pois precisamos passar o AuthService para o UsersController como dependência. Vamos começar alterando o arquivo de rota onda o UsersController é construído adicionando a dependência ao construtor.

```
import express from 'express';
import UsersController from '../controllers/users';
import User from '../models/user';
+import AuthService from '../services/auth';

const router = express.Router();
- const usersController = new UsersController(User);
+ const usersController = new UsersController(User, AuthService);
```

Próximo é passo alterar o Users Controller:

```
class UsersController {
- constructor(User) {
+  constructor(User, AuthService) {
    this.User = User;
+    this.AuthService = AuthService;
  };
```

Agora vamos alterar o metodo authenticate para utilizar o AuthService, o método deve ficar assim:

```
  async authenticate(req, res) {
    const authService = new this.AuthService(this.User);
    const user = await authService.authenticate(req.body);
    if(!user) {
      return res.sendStatus(401);
    }
    const token = jwt.sign({
      name: user.name,
      email: user.email,
      password: user.password,
      role: user.role
    }, config.get('auth.key'), {
      expiresIn: config.get('auth.tokenExpiresIn')
    });
    return res.send({ token });
  }
```

Se executarmos os testes de unidade os testes do método authenticate do users controller estarão quebrando, pois mudamos o código, antes de qualquer alteração nesse teste vamos executar os testes de integração para garantir que a resposta final ainda é a mesma. Se executarmos o teste de integração agora vamos receber um erro do nacl dizendo que não temos permissão para fazer a request, nesse momento entramos em um dilema clássico no mundo dos testes onde é necessário

fazer mais de uma alteração para conseguir seguir em frente. Para que os nossos testes de integração passem precisamos gerar um JWT e adicionar na request, já possuímos um AuthService então vamos adicionar essa lógica lá para que seja reutilizável no futuro. Sempre começando pelo teste, vamos adicionar o seguinte no teste de unidade do AuthService:

```
import AuthService from '../../../src/services/auth';
import bcrypt from 'bcrypt';
import Util from 'util';
import sinon from 'sinon';
+ import jwt from 'jsonwebtoken';
+ import config from 'config';
```

Abaixo do contexto do authenticate adicione o seguinte caso de teste:

```
  context('generateToken', () => {
    it('should generate a JWT token from a payload', () => {
      const payload = {
        name: 'John',
        email: 'jhondoe@mail.com',
        password: '12345'
      };
      const expectedToken =  jwt.sign(payload, config.get('auth.key'), {
        expiresIn: config.get('auth.tokenExpiresIn')
      });
      const generatedToken = AuthService.generateToken(payload);
      expect(generatedToken).to.eql(expectedToken);
    });
  });
```

Nada de novo aqui, essa á a mesma lógica utilizada para gerar o token no authenticate do UsersController, no futuro vamos refatorar este método para usar o generateToken do AuthService também. Executando os testes de unidade agora eles estão quebrando, afinal esse método ainda não existe no AuthService, vamos criá-lo agora:

```
import bcrypt from 'bcrypt';
+ import jwt from 'jsonwebtoken';
+ import config from 'config';
```

Adicione o seguinte método, logo abaixo do método authenticate.

```
  static generateToken(payload) {
    return jwt.sign(payload, config.get('auth.key'), {
      expiresIn: config.get('auth.tokenExpiresIn')
    });
  }
```

Agora os testes de unidade para o AuthService devem estar passando (os testes do UsersController vão estar falhando e isso é esperado):

```
  Service: Auth
    generateToken
      ✓ should generate a JWT token from a payload
```

Agora vamos adicionar a lógica de gerar token aos testes de integração de user:

```
import User from '../../../src/models/user';
+ import AuthService from '../../../src/services/auth';
```

Adicione a seguinte linha às definições de constants:

```
 const expectedAdminUser = {
    _id: defaultId,
    name: 'Jhon Doe',
    email: 'jhon@mail.com',
    role: 'admin'
  };
 + const authToken = AuthService.generateToken(expectedAdminUser);
```

Aqui estamos gerando um JWT manualmente, ele está sendo adicionado às requisições logo abaixo.

```
 request
        .get('/users')
+        .set({'x-access-token': authToken})

        request
          .get(`/users/${defaultId}`)
+          .set({'x-access-token': authToken})

        request
          .post('/users')
+          .set({'x-access-token': authToken})

```

```
13          request
14            .put(`/users/${defaultId}`)
15  +          .set({'x-access-token': authToken})
16
17
18          request
19            .delete(`/users/${defaultId}`)
20  +          .set({'x-access-token': authToken})
```

Depois dessa alteração os testes de integração de users devem estar passando:

```
1   Routes: Products
2     GET /products
3       1) should return a list of products
4       when an id is specified
5         2) should return 200 with one product
6     POST /products
7       when posting a product
8         3) should return a new product with status code 201
9     PUT /products/:id
10      when editing a product
11        ✓ should update the product and return 200 as status code
12    DELETE /products/:id
13      when deleting a product
14        4) should delete a product and return 204 as status code
15
16   Routes: Users
17    GET /users
18      ✓ should return a list of users
19      when an id is specified
20        ✓ should return 200 with one user
21    POST /users
22      when posting an user
23        ✓ should return a new user with status code 201
24    PUT /users/:id
25      when editing an user
26        ✓ should update the user and return 200 as status code
27    DELETE /users/:id
28      when deleting an user
29        ✓ should delete an user and return 204 as status code
30    when authenticating an user
31      ✓ should generate a valid token
32      ✓ should return unauthorized when the password does not match
```

```
33
34
35    8 passing (994ms)
36    4 failing
```

Não vamos nos preocupar com products agora, o que queremos ver conseguimos, o teste end 2 end está funcionando e isso significa que o users controller está com a lógica certa, podemos alterar o teste de unidade do método authenticate agora para que volte a passar, os testes devem ficar assim:

```
1  describe('authenticate', () => {
2      it('should authenticate a user', async () => {
3        const fakeUserModel = {};
4        const user = {
5          name: 'Jhon Doe',
6          email: 'jhondoe@mail.com',
7          password: '12345',
8          role: 'admin'
9        };
10       const userWithEncryptedPassword = {
11         ...user,
12         password: bcrypt.hashSync(user.password, 10)
13       };
14       class FakeAuthService {
15         authenticate() {
16           return Promise.resolve(userWithEncryptedPassword)
17         }
18       };
19
20       const jwtToken = jwt.sign(
21         userWithEncryptedPassword,
22         config.get('auth.key'),
23         {
24           expiresIn: config.get('auth.tokenExpiresIn')
25         }
26       );
27       const fakeReq = {
28         body: user
29       };
30       const fakeRes = {
31         send: sinon.spy()
32       };
33       const usersController = new UsersController(fakeUserModel, FakeAuthService);
34       await usersController.authenticate(fakeReq, fakeRes);
```

```
35        sinon.assert.calledWith(fakeRes.send, { token: jwtToken });
36      });
37
38
39      it('should return 401 when the user can not be found', async () => {
40        const fakeUserModel = {};
41        class FakeAuthService {
42          authenticate() {
43            return Promise.resolve(false)
44          }
45        };
46        const user = {
47          name: 'Jhon Doe',
48          email: 'jhondoe@mail.com',
49          password: '12345',
50          role: 'admin'
51        };
52        const fakeReq = {
53          body: user
54        };
55        const fakeRes = {
56          sendStatus: sinon.spy()
57        };
58        const usersController = new UsersController(fakeUserModel, FakeAuthService);
59
60        await usersController.authenticate(fakeReq, fakeRes);
61        sinon.assert.calledWith(fakeRes.sendStatus, 401);
62      });
```

O caso de teste “should return 401 when the password does not match” precisa ser removido pois ele nao está ciente da adição do AuthService, no mundo do TDD para fazer esse tipo de alteração precisamos estar seguros. Em nosso caso a maneira mais simples de ficarmos seguros é testar a lógica que compara as senhas no AuthService.

Os testes de unidade devem estar passando, isso que significa que o AuthService está tratando esse caso, agora estamos seguros para remover o caso de teste do users controller.

```
-    it('should return 401 when the password does not match', async () => {
-      const fakeUserModel = {
-        findOne: sinon.stub()
-      };
-      const user = {
-        name: 'Jhon Doe',
-        email: 'jhondoe@mail.com',
-        password: '12345',
-        role: 'admin'
-      };
-      const userWithDifferentPassword = {
-        ...user,
-        password: bcrypt.hashSync('another_password', 10)
-      };
-      fakeUserModel.findOne.withArgs({ email: user.email }).resolves({
-        ...userWithDifferentPassword
-      });
-      const fakeReq = {
-        body: user
-      };
-      const fakeRes = {
-        sendStatus: sinon.spy()
-      };
-      const usersController = new UsersController(fakeUserModel);
-
-      await usersController.authenticate(fakeReq, fakeRes);
-      sinon.assert.calledWith(fakeRes.sendStatus, 401);
-    });
```

Precisamos alterar o método authenticate que vai utilizar o AuthService. Vamos começar alterando o teste should authenticate a user:

```
      const userWithEncryptedPassword = {
        ...user,
        password: bcrypt.hashSync(user.password, 10)
      };
+     const jwtToken = jwt.sign(userWithEncryptedPassword,
+       config.get('auth.key'),{
+         expiresIn: config.get('auth.tokenExpiresIn')
+       });

      class FakeAuthService {
        authenticate() {
```

```
12            return Promise.resolve(userWithEncryptedPassword)
13          }
14
15  +       static generateToken() {
16  +         return jwtToken;
17  +       }
18        };
19
20  -     const jwtToken = jwt.sign(
21  -       Object.assign({}, user, { password: hashedPassword }),
22  -       config.get('auth.key'),{
23  -         expiresIn: config.get('auth.tokenExpiresIn')
24  -       });
25  -
```

No código acima adicionamos um método fake para simular o generateToken do AuthService que retorna um token manualmente gerado. Se executarmos os testes de unidade agora eles vão estar quebrando, precisamos alterar o método authenticate para usar o generateToken no UsersController:

```
1   -       const token = jwt.sign({
2   +       const token = this.AuthService.generateToken({
3             name: user.name,
4             email: user.email,
5             password: user.password,
6             role: user.role
7   -       }, config.get('auth.key'), {
8   -           expiresIn: config.get('auth.tokenExpiresIn')
9   -         });
10  +       });
```

Vamos remover também as dependências que não serão mais necessárias:

```
1  -import jwt from 'jsonwebtoken';
2  -import config from 'config';
3  -import bcrypt from 'bcrypt';
```

Os testes de unidade devem estar passando.

Último passo é fixar os testes end 2 end da rota de products, vamos alterar o products_spec.js:

```
import Product from '../../../src/models/product';
+ import AuthService from '../../../src/services/auth';
```

Agora vamos adicionar o expectedAdminUser para poder gerar o token, e a seguir o código para gerar o token

```
 const expectedProduct = {
    __v: 0,
    _id: defaultId,
    name: 'Default product',
    description: 'product description',
    price: 100
  };
+        const expectedAdminUser = {
+   _id: defaultId,
+           name: 'Jhon Doe',
+   email: 'jhon@mail.com',
+   role: 'admin'
+  };
+  const authToken = AuthService.generateToken(expectedAdminUser);
```

O próximo passo será adicionar o authToken nas requests:

```
request
        .get('/products')
+        .set({'x-access-token': authToken})
        .end((err, res) => {
          expect(res.body).to.eql([expectedProduct]);
          done(err);


       request
          .get(`/products/${defaultId}`)
+         .set({'x-access-token': authToken})
          .end((err, res) => {
            expect(res.statusCode).to.eql(200);
            expect(res.body).to.eql([expectedProduct]);
        request
          .post('/products')
+               .set({'x-access-token': authToken})
          .send(newProduct)
          .end((err, res) => {
```

```
20              expect(res.statusCode).to.eql(201);
21
22
23          request
24            .put(`/products/${defaultId}`)
25  +         .set({'x-access-token': authToken})
26            .send(updatedProduct)
27            .end((err, res) => {
28              expect(res.status).to.eql(200);
29
30          request
31            .delete(`/products/${defaultId}`)
32  +          .set({'x-access-token': authToken})
33            .end((err, res) => {
34              expect(res.status).to.eql(204);
35              done(err);
```

Pronto! Todos os testes devem estar passando

```
1   $ npm test
2
3    Controller: Products
4       get() products
5         ✓ should call send with a list of products
6         ✓ should return 400 when an error occurs
7       getById()
8         ✓ should call send with one product
9       create() product
10        ✓ should call send with a new product
11        when an error occurs
12          ✓ should return 422
13      update() product
14        ✓ should respond with 200 when the product has been updated
15        when an error occurs
16          ✓ should return 422
17      delete() product
18        ✓ should respond with 204 when the product has been deleted
19        when an error occurs
20          ✓ should return 400
21
22     Controller: Users
23      get() users
24        ✓ should call send with a list of users
```

```
      ✓ should return 400 when an error occurs
    getById()
      ✓ should call send with one user
    create() user
      ✓ should call send with a new user
      when an error occurs
        ✓ should return 422
    update() user
      ✓ should respond with 200 when the user has been updated
      when an error occurs
        ✓ should return 422
    delete() user
      ✓ should respond with 204 when the user has been deleted
      when an error occurs
        ✓ should return 400
    authenticate
      ✓ should authenticate a user
      ✓ should return 401 when theres no user

  AuthMiddleware
    ✓ should verify a JWT token and call the next middleware
    ✓ should call the next middleware passing an error when the token validation fai\
ls
    ✓ should call next middleware if theres no token

  Service: Auth
    authenticate
      ✓ should authenticate a user
      ✓ should return false when the password does not match
    generateToken
      ✓ should generate a JWT token from a payload


  26 passing (259ms)


> node-book@1.0.0 test:integration /Users/wneto/Dev/building-testable-apis-with-node\
js-code
> NODE_ENV=test mocha --opts test/integration/mocha.opts test/integration/**/*_spec.\
js

  Routes: Products
    GET /products
```

```
      ✓ should return a list of products
      when an id is specified
        ✓ should return 200 with one product
    POST /products
      when posting a product
        ✓ should return a new product with status code 201
    PUT /products/:id
      when editing a product
        ✓ should update the product and return 200 as status code
    DELETE /products/:id
      when deleting a product
        ✓ should delete a product and return 204 as status code

  Routes: Users
    GET /users
      ✓ should return a list of users
      when an id is specified
        ✓ should return 200 with one user
    POST /users
      when posting an user
        ✓ should return a new user with status code 201
    PUT /users/:id
      when editing an user
        ✓ should update the user and return 200 as status code
    DELETE /users/:id
      when deleting an user
        ✓ should delete an user and return 204 as status code
    when authenticating an user
      ✓ should generate a valid token
      ✓ should return unauthorized when the password does not match


  12 passing (1s)
```

Código deste capitulo está aqui no github[93]

[93] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step11

Estilo de código e formatação

Até o momento viemos desenvolvendo sem prestar muita atenção no estilo do código. Agora que estamos na reta final é hora de adicionar algumas ferramentas para garantir que o nosso código vai ter o mesmo estilo e formatação independente da pessoa ou da IDE que está sendo utilizada. Existem várias ferramentas na comunidade para fazer esse serviço, aqui vamos utilizar eslint e prettier.

Eslint

O eslint[94] é uma ferramenta utilizada para identificar e reportar erros de padrões de código como sintaxe e estilo em Javascript e é apoiada por grandes empresas como Airbnb e Facebook.

Prettier

O prettier[95] é um formatador de código que já vem praticamente configurado, basta instalar e começar a utilizar, ele segue os padrões mais comuns de formatação de código Javascript da comunidade.

Com a combinação de eslint e prettier é póssivel automatizar o trabalho de análise estática de código acelerando assim processos como o de revisão de pull requests.

Configuração

```
$ npm install --save-dev eslint@^6.7.2 babel-eslint@^10.0.3 eslint-plugin-node@^10.0\
.0 prettier@^1.19.1
```

Adicione os seguintes comandos ao package.json

```
    "lint": "eslint src --ext .js",
    "lint:fix": "eslint src --fix --ext .js",
    "prettier:list": "prettier --check 'src/**/*.js'",
    "prettier:fix": "prettier --write 'src/**/*.js'",
    "style:fix": "npm run lint:fix & npm run prettier:fix"
```

[94] https://eslint.org/
[95] https://prettier.io/

Aqui descrevemos comandos para analisar e fixar os problemas, é importante ter comandos separados pois quando executamos a análise em uma ferramenta de CI queremos que o build apenas quebre, se o comando tentar fixar os arquivos ele vai ficar somente no CI mas não na branch. O correto é quebrar o build e o desenvolvedor arrumar e pushar o código arrumado novamente para a branch.

No nosso ambiente de desenvolvimento podemos executar diretamente os comandos de fix, as ferramentas tentam arrumar o máximo possível automaticamente mas algumas coisas necessitam da revisão manual.

Fixando os problemas de estilo e formatação

Como falei, no ambiente de desenvolvimento podemos executar diretamente o comando de fix, vamos começar com o eslint:

```
$npm run lint:fix
```

Minha saída foi a seguinte

```
/Users/wneto/Dev/building-testable-apis-with-nodejs-code/src/controllers/users.js
  1:8  error  'jwt' is defined but never used     no-unused-vars
  2:8  error  'config' is defined but never used  no-unused-vars
  3:8  error  'bcrypt' is defined but never used  no-unused-vars

/Users/wneto/Dev/building-testable-apis-with-nodejs-code/src/models/user.js
  21:10  error  'err' is not defined                no-undef
  26:25  error  'options' is defined but never used  no-unused-vars
```

Estes são os arquivos que não puderam ser corrigidos automaticamente, basta ir neles e corrigir manualmente. Por exemplo o arquivo *src/controllers/users.js* possui imports que não estão sendo utilizados e o *src/models/user.js* possui um erro de digitação que fazia o err ser undefined, agora percebemos o valor da ferramenta, não é? Provavelmente notaríamos isso somente em produção, quando um erro ocorresse.

Todo o código atualizado por ser encontrado aqui[96]

[96] https://github.com/waldemarnt/building-testable-apis-with-nodejs-code/tree/step12

Final

Este capítulo encerra o livro. Agora você deve possuir o conhecimento e as ferramentas necessárias para começar a construir suas próprias APIs escaláveis com Node.js seguindo os melhores padrões de qualidade. Vamos continuar essa conversa no youtube[97], onde vou seguir publicando conteúdo atualizado.

Obrigado pela jornada!

[97] https://www.youtube.com/user/waldemaneto