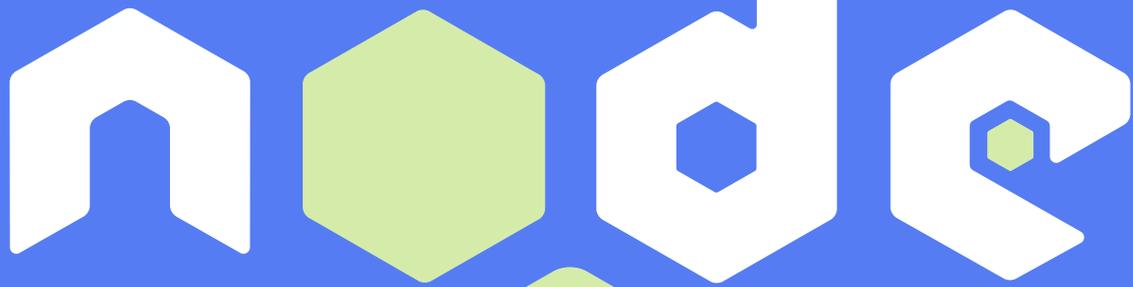


Boas Práticas com



Umbler



Sumário

Clique nos capítulos
para acessá-los

1. INTRODUÇÃO	3
2. BONS HÁBITOS E PRINCÍPIOS GERAIS	6
Use Node para os projetos certos	7
Node.js foi criado para async	8
Ajude seu cérebro a entender o projeto	8
Crie apenas os diretórios que precisa	8
Torne fácil localizar arquivos de código	9
Diminua o acoplamento e aumente o reuso de software	9
Priorize configuration over convention	9
Nomes de arquivos no formato 'lower-kebab-case'	10
Tenha um padrão de código a.k.a. Guia de Estilo	10
3. CODIFICANDO	12
Comece todos os projetos com npm init	13
Entenda o versionamento de pacotes	13
Configure scripts no package.json	15
Use variáveis de ambiente	16
Carregue suas dependências antecipadamente	18
Centralize a criação de conexões ao banco	19
Mindset funcional	19
JS Full-stack	21
4. TESTANDO	24
Use uma boa ferramenta	25
Trate bem os seus erros	26
Aumente a visibilidade dos erros	26
Vendo os objetos completamente no console	27
Escreva testes unitários	28
5. DEPLOY	29
Não use FTP!	30
Cuide da segurança	31
Mantendo seu processo no ar!	31
Trabalhando com vários cores	34
Use Integração/entrega/implantação contínua	35
6. MÓDULOS RECOMENDADOS	38
7. SEGUINDO EM FRENTE	42

INTRODUÇÃO



Without requirements and design, programming is the art of adding bugs to an empty text file.

Louis Srygley



Sabe quando você faz um trabalho e ele fica tão bom que você pensa algo como: **“eu gostaria de ter tido acesso a isso quando estava começando”**?

Pois é, esse é o meu sentimento com este ebook.

Quando estamos começando com uma nova tecnologia sempre tudo é muito confuso, complexo e, por que não dizer, assustador?

Trabalho há onze anos com programação e há uns treze programas mesmo que não profissionalmente, desde um técnico em eletrônica que fiz em 2004 que incluía programação de microcontroladores para a indústria. Nesse ínterim tive de trocar sucessivas vezes de tecnologias, seja por questões acadêmicas, seja por questões de mercado: Assembly, C/C++, Lisp, Prolog, Visual Basic, Java, Lua, .NET, PHP e, agora, **Node.js**.

E a cada troca eu volto a ser um calouro, tendo de aprender não apenas sintaxe, semântica, todo o “ferramental” relacionado à tecnologia, mas o mais difícil de tudo: **boas práticas!**

Como saber se o código que estou escrevendo está dentro dos padrões de mercado?

Como tirar o máximo proveito da linguagem através do uso das melhores ferramentas e extensões?

Como é a arquitetura ou o mindset correto para os tipos de projetos que são criados com esta tecnologia?

Como ...

E tudo fica pior quando estamos falando de uma tecnologia que não tem sequer uma década de vida ainda e até cinco anos atrás era coisa de devs malucos em eventos. São muitas as dúvidas sobre Node.js e, mesmo com o todo-poderoso Google em nossas mãos, parece que essas respostas estão desconectadas e jogadas cada uma em um canto obscuro da Internet, fazendo com que dê um trabalho enorme encontrá-las.

Não trago aqui verdades universais ou respostas para todas as perguntas que você vai ter sobre Node.js, mas trago um apanhado da experiência coletiva de milhares de desenvolvedores ao redor do globo para formar um guia de boas práticas com Node.js. Pessoas com muito mais conhecimento de Node do que eu, e algumas dicas minhas também, afinal eu também sou um desses milhares de desenvolvedores!

Talvez inclusive você reconheça alguma dica que você mesmo pode ter dado em alguma rede social ou publicação na Internet.

Espero que goste do que preparamos pra você e que este ebook seja útil na sua jornada como programador Node.js. Este é o grande propósito da Umblar:

 **Facilitar o desenvolvimento web, seja com a nossa plataforma, seja com materiais como esse.**

Ah, e se um dia o seu projeto virar uma startup de sucesso e você for fazer um IPO ou receber uma proposta de compra vinda do Facebook, lembra de quem te ajudou lááááá no início. ;)

Um abraço e sucesso.



Luiz Duarte
Dev Evangelist
Umblar

BONS HÁBITOS E PRINCÍPIOS GERAIS



I'm not a great programmer;
I'm just a good programmer
with great habits.

Kent Beck



Acima de qualquer boa prática que eu possa propor neste ebook, vale ressaltar alguns princípios e motivações que devem estar acima de qualquer coisa em seus projetos com Node.js.

Nem todos princípios vão se encaixar apenas no Node.js e talvez você reconheça uma ou outra coisa que já utilize hoje em outra tecnologia, como PHP. Ótimo!

Por outro lado, talvez encontre críticas a hábitos ruins que você possui hoje. Não se assuste, parte do processo de evolução como programadores (e como pessoas!) envolve manter bons hábitos e evitar hábitos ruins o tempo todo.

Afinal, é como dizem, “o hábito faz o monge”.



Use Node para os projetos certos

Existe um dito popular que diz: “Nem todo problema é um prego e nem toda ferramenta um martelo”.

Qualquer tecnologia que lhe vendam como sendo a solução para todos seus problemas é no mínimo para se desconfiar. Node não é um canivete-suíço e possui um propósito de existir muito claro desde sua homepage até a seção de about no site oficial: ele te proporciona uma plataforma orientada à eventos, assíncrona e focada em I/O e aplicações de rede não-bloqueantes.

Isso não quer dizer que você não possa fazer coisas com ele para as quais ele não foi originalmente concebido, apenas que ele pode não ser a melhor ferramenta para resolver o problema e você pode acabar frustrado achando que a culpa é do Node. Por isso, use Node preferencialmente para:

- APIs;
- Bots;
- Mensageria;
- IoT;
- Aplicações real-time;

Node.js foi criado para async

Isso pode parecer óbvio quando falamos de uma plataforma focada em I/O não-bloqueante, mas parece que tem programador que a todo custo quer mudar isso em suas aplicações, fazendo todo tipo de malabarismo para transformar trechos de código e módulos originalmente assíncronos em tarefas síncronas.

Esqueça. Nem perca seu tempo.

Se não gosta de callbacks (será que alguém gosta?) use promises e async/await, mas não tente transformar Node.js em uma plataforma síncrona pois esse não é o objetivo dele e ele não funciona bem dessa forma, afinal ele é single-thread, lembra? Se realmente precisa que tudo rode sincronamente, use PHP ou outra plataforma que crie threads separadas e seja feliz.



Ajude seu cérebro a entender o projeto

Conseguimos lidar com apenas pequenas quantidades de informação de cada vez. Alguns cientistas falam de sete coisas ao mesmo tempo, outros falam de quatro coisas. Por isso que usamos pastas, módulos e por isso que criamos funções. Elas nos ajudam a lidar com a complexidade do sistema nos permitindo mexer em pequenas porções dele de cada vez.



Crie apenas os diretórios que precisa

Não saia criando dezenas de diretórios que acabarão vazios ou com apenas um arquivo dentro. Comece com o básico de pastas que você precisa e vá adicionando conforme a complexidade for aumentando. Afinal, você não compra um ônibus como primeiro veículo pensando no dia em que pode querer dar carona para muita gente, certo?

Evite o over engineering de querer ter a melhor arquitetura possível no “dia um” do seu projeto. Você deve conhecer (e evitar) o termo ‘código espaguete’, certo? Um projeto com mais pastas do que o necessário é tão nocivo quanto, é o chamado ‘código lasanha’ (muitas camadas)!

Torne fácil localizar arquivos de código

Os nomes de pasta devem ser significativos, óbvios e fáceis de entender. Código que não é mais usado, bem como arquivos antigos, devem ser removidos do projeto, e não apenas comentados ou renomeados. Afinal, quem gosta de coisas antigas e que não são mais usadas é museu, certo?



Diminua o acoplamento e aumente o reuso de software.

A sugestão aqui é ter mais módulos especialistas na sua aplicação o mais desacoplados possível, até mesmo em projetos separados registrados no NPM, por exemplo. Uma vez que você tenha módulos independentes, sua aplicação fica menos acoplada e a manutenção mais simples.

Uma dica para aumentar o reuso de software mantendo o acoplamento baixo é não colocar código “especializado” nas suas chamadas de model e controller. Por exemplo, se após salvar um cliente no banco você precisa enviar um email pra ele, não coloque o código que envia email logo após o salvamento no banco, ao invés disso, dispare uma função de um módulo de email.

Outra dica é programar com um mindset mais funcional, conforme citado no capítulo sobre codificação.

Uma terceira e última dica é aproveitar os módulos já existentes, sempre procurando a solução para o que você precisa antes de começar a desenvolver ela. A lista de módulos recomendados no capítulo final é um bom começo, mas use e abuse do site do NPM e do StackOverflow para isso.



Priorize configuration over convention

Não faça coisas mágicas como assumir que o nome de um arquivo é x ou que os uploads vão parar sempre em uma pasta y. Esse hábito é chamado popularmente de convention over configuration. Isso é frágil e facilmente quebra sua aplicação quando alguma coisa muda e você não lembra de todas as convenções que criou

para sua arquitetura. Programe de maneira que o código em si possa fazer com que o programador entenda todo o projeto.

Use variáveis de ambiente (como será citado mais pra frente) e/ou arquivos JSON de configuração, o que preferir, mas configure o seu projeto adequadamente e de preferência de maneira independente de infraestrutura, para que você consiga subir sua aplicação rapidamente em qualquer lugar.



Nomes de arquivos no formato 'lower-kebab-case'

Use apenas minúsculas, sem acentos e '-' como separador entre palavras. Isso evita problemas de sistemas de arquivos em sistemas operacionais diferentes que sua aplicação Node possa rodar. Ex: cadastro-cliente.js

Este é o padrão do NPM para arquivos e pastas, então é melhor seguir esta regra, por mais que não goste dela.

Veja mais regras adiante.



Tenha um padrão de código a.k.a. Guia de Estilo

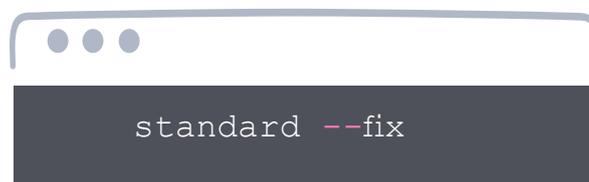
Muitos programadores que conheço não gostam de padrões de código. No entanto, um ponto importante a se entender é: padrões de código não são para o programador, mas sim para a empresa ou para a comunidade (no caso dos projetos open-source).

Uma vez que em empresas e projetos abertos existe uma certa rotatividade de profissionais trabalhando no mesmo código, é natural que programadores com diferente visões de como o software deve ser escrito podem mexer nele e fazer com que facilmente não exista coerência na escrita. Padrões evitam isso.

Por mais que alguns podem achar isso preciosismo, só quem já trabalhou em grandes bases e código sem padrão que sabe o quão difícil é se achar no meio de um projeto bagunçado. Para resolver isso, empresas e projetos de todos os tipos e tamanhos decidem padrões. Um ponto importante aqui é que não é o mesmo padrão para todos(as). A regra é clara: pegue um padrão que você se sinta confortável e que funcione na sua empresa/projeto.

Aqui vai uma sugestão que está ganhando cada vez mais força: StandardJS, disponível em <http://standardjs.com>. É um padrão rígido, inflexível, construído com opiniões fortes mas que rapidamente está dominando diversos projetos e empresas mundo afora. Eu não concordo com tudo o que prega o StandardJS, mas que ajuda a tornar o código muito mais legível e enxuto, isso com certeza.

Para ajudar com projetos já existentes, ele possui um fixer automático que corrige a maior parte dos códigos fora do estilo e, dependendo da ferramenta que estiver usando (eu uso o Visual Studio Code), te dá sugestões real-time do que você está fazendo “errado”. Costumo trabalhar em projetos remotos com outros desenvolvedores e é uma excelente maneira de garantir uma padronização entre os códigos escritos por todos. Basta rodar:

A terminal window with a dark background and light text. The command 'standard --fix' is entered. The window has three small circles in the top-left corner representing window controls.

```
standard --fix
```

e praticamente tudo se resolve!

E se eu não consegui te convencer, a página deles está cheia de empresas/projetos que usem esse estilo, como: NPM, GitHub, ZenDesk, MongoDB, Typeform, ExpressJS e muito mais. Falando de NPM especificamente, muitos, mas muitos, dos mais populares módulos existentes foram escritos usando StandardJS como guia de estilo, entre eles o mocha, dotenv, mssql, express-generator e muito mais.

CODIFICANDO



Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler



Node.js se tornou uma das plataformas mais populares nos últimos anos. Parte desse sucesso se deve ao fato de que é muito fácil começar projetos em Node.js, mas uma vez que você vá além do Olá Mundo básico, saber como codificar corretamente sua aplicação e como lidar com tantos erros pode facilmente se tornar um pesadelo (aliás, como na maioria das linguagens de programação).

Infelizmente, evitar que este pesadelo se torne realidade faz toda a diferença entre uma aplicação sólida em produção e um desastre em forma de software.

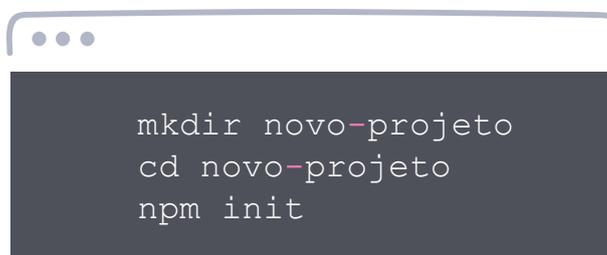
Com isso em mente, vamos passar neste capítulo por uma série de boas práticas usando Node.js que te ajudarão a evitar a maioria das armadilhas desta plataforma.



Comece todos os projetos com npm init

Vamos começar do básico, ok?

Todos estamos acostumados a usar o NPM para instalar novos pacotes em nosso projeto, mas sua utilidade vai além disso. Primeiramente, eu recomendo que comece todos os seus projetos usando npm init, como abaixo:



```
mkdir novo-projeto
cd novo-projeto
npm init
```

Isto faz com que o **package.json** seja criado, o que permite que você adicione diversos metadados que ajudarão você e sua equipe mais tarde,, definindo a versão do node que roda o projeto, quais dependências ele possui, qual é o comando para inicializar o projeto...opa essa é uma dica quente, vamos explorá-la?



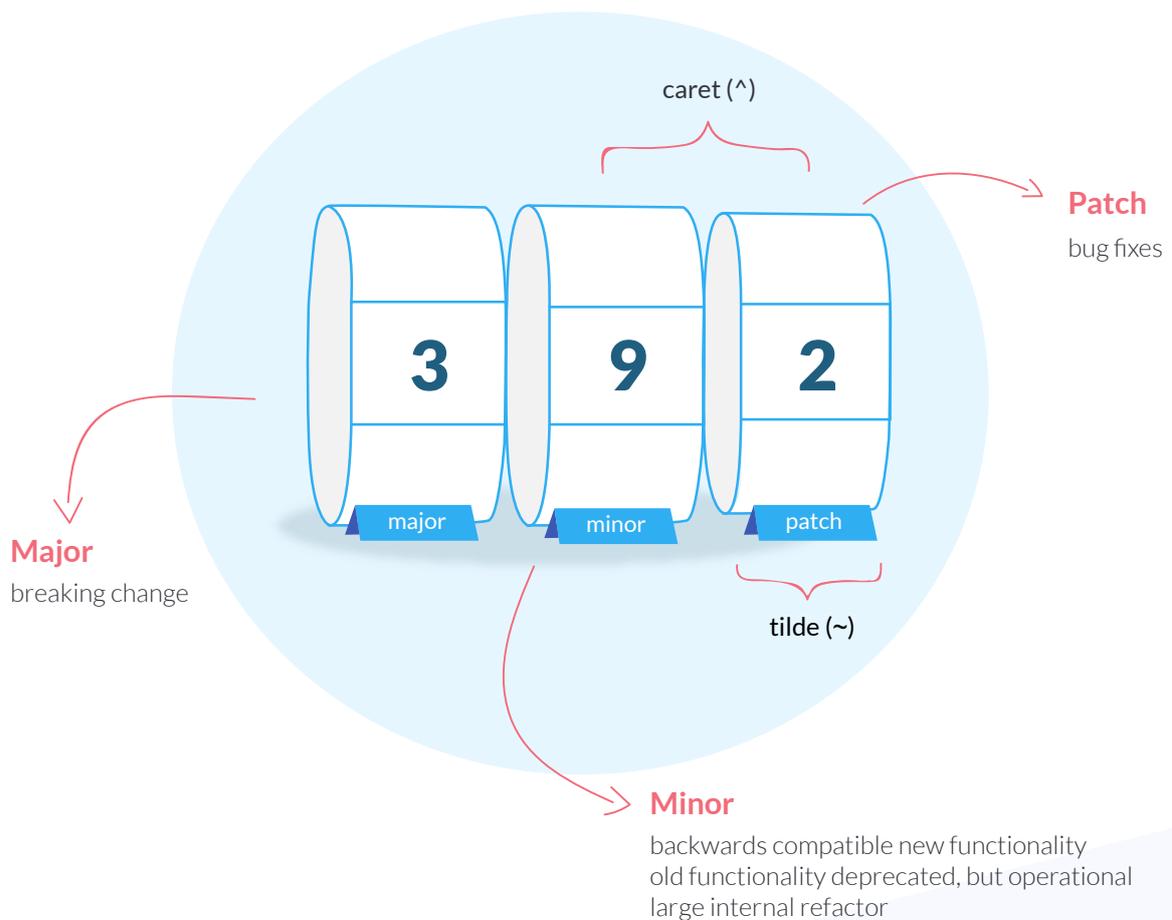
Entenda o versionamento de pacotes

Então você abre o package.json e rapidamente entende que as dependencies são os pacotes que sua aplicação usa, mas onde deveriam estar listadas as versões dos pacotes tem um monte de símbolos que não dizem muita coisa.

Resumidamente funciona assim:

- O til garante que o pacote seja sempre carregado respeitando o número do meio da versão. Ex: `~1.2.3` pega o pacote mais recente da versão 1.2.x, mas não vai atualizar para 1.3. Geralmente garante que correções de bugs sejam atualizados no seu pacote.
- O circunflexo garante que o pacote seja sempre carregado respeitando o primeiro número da versão. Ex: `^1.2.3` pega o pacote mais recente da versão 1.x, mas não vai atualizar para 2.0. Garante que bugs e novas funcionalidades do seu pacote sejam atualizados, mas não novas versões “major” dele.

A imagem abaixo ajuda a entender o template de versões dos pacotes do NPM, que aliás usa um padrão bem comum da indústria de software:



Outros símbolos incluem:

- `>`, `>=`, `<`, `<=1.0`: a versão deve ser superior, superior ou igual, inferior, inferior ou igual à 1.0, respectivamente.
- `1.2.x`: equivalente a `~1.2.0`
- `*`: qualquer versão do pacote
- `latest`: a versão mais recente do pacote

Se você não tiver símbolo algum, aí o pacote deve ser sempre carregado usando a versão especificada.

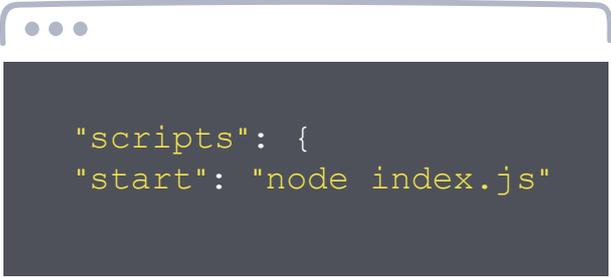
Uma última dica bem valiosa para quando se quer atualizar todos pacotes é colocar `*` na versão de todos e rodar o comando `npm update --save` sobre a pasta do projeto.



Configure scripts no package.json

Lembra da dica de priorizar “configuration over convention” que mencionei lá atrás? Pois é, em diversas ocasiões temos de rodar scripts para fazer tarefas em nossas aplicações e se tivermos de lembrar qual script rodar, em qual ordem, em qual situação, etc certamente uma hora alguém vai fazer alguma besteira por esquecimento.

Uma constante são os scripts de inicialização, mas existem outros, todos podendo ser configurados na propriedade `scripts` do `package.json`, como abaixo:



```
"scripts": {  
  "start": "node index.js"  
}
```

Neste caso, quando executarmos o comando `npm start` dentro da pasta deste projeto, ele irá executar `node index.js`, sem que tenhamos de escrever este comando manualmente. Ok, aqui foi um exemplo de comando simples, mas poderia ser mais complicado, como passando flags e até mesmo variáveis de ambiente.

Além do script “start” para inicialização, temos:

- test: para rodar os seus testes automaticamente (útil para facilitar CI);
- preinstall: para executar alguma coisa logo que um “npm install” é chamado, antes da instalação realmente acontecer;
- postinstall: para executar alguma coisa depois que um “npm install” acontecer;

E por fim, caso queira scripts personalizados, você pode adicionar o nome que quiser nos scripts, mas para chamá-lo depois deve usar o comando “npm run-script nomeDoScript”, facilitando a sua vida caso seu time tenha vários scripts diferentes.



Use variáveis de ambiente

Uma das premissas centrais do Node.js é o desacoplamento. Mais tarde falaremos de micro serviços (que é uma excelente prática para Node), mas, por ora, entenda que um dos acoplamentos que mais tomam tempo de gerência de configuração são os relacionados à serviços externos como bases de dados, APIs, etc.

Isso porque esses serviços externos possuem diferentes configurações nos diferentes ambientes que sua aplicação Node vai passar: configs de produção, de teste, de homologação, na sua máquina, etc. Como lidar com isso sem que os desenvolvedores tenham de conhecer as configurações de produção, sem ferrar com seu CI, etc? Com variáveis de ambiente!

Sempre que subimos uma aplicação Node podemos passar à ela as variáveis de ambiente daquele processo (process.env) antes do comando de inicialização:

```
NOME_VARIAVEL=valor node index.js
```

ou em Windows

```
SET NOME_VARIAVEL=valor node index.js
```

Assim, podemos passar configurações locais como variáveis de ambiente para aquela sessão da aplicação, evitando ter connection strings, por exemplo, hard-coded na sua aplicação.

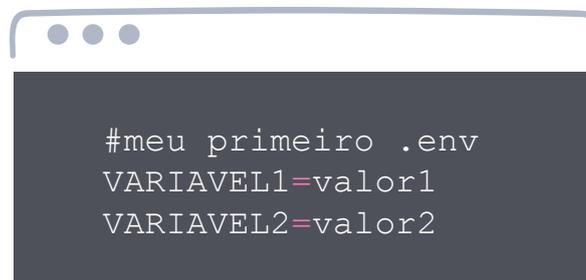
Se você não achou prático, saiba que eu também não gosto de digitar isso aí toda vez que vou executar uma aplicação Node. Sendo assim, uma maneira mais profissional seria usar essa dica em conjunto com a dica anterior, de configurar scripts no package.json. Basta que você adicione as variáveis de ambiente no comando de start, como abaixo:



```
"scripts": {  
  "start": "set NOME_VARIAVEL=valor && node app"  
}
```

No entanto, se seu package.json estiver versionado, e geralmente está, isso pode não ser uma boa, porque ao fazer commit na master ele pode ser enviado ao servidor (se estiver usando CI) e estragar sua aplicação de produção. Neste caso, sugiro uma abordagem ainda mais profissional: arquivos .env.

Alguns desenvolvedores criam arquivos de configuração próprios que são lidos na inicialização da aplicação para carregar estas variáveis, no entanto, arquivos .env e o módulo dotenv (e a versão “segura” dotenv-safe) resolvem este problema facilmente. Primeiro, crie o seu arquivo .env (sem nome, apenas “.env” mesmo) e coloque uma variável de ambiente por linha, como abaixo, podendo usar # para comentários:



```
#meu primeiro .env  
VARIABLE1=valor1  
VARIABLE2=valor2
```

Agora adicione no gitignore este arquivo, para ele não ser versionado. Certifique-se de fazer cópias dele nos seus diversos ambientes com os valores de teste, homologação, produção, etc, assim, você nunca envia ele e nunca sobrescreve as configs de produção.

Para carregá-lo, é muito simples, chame esta linha de código na primeira linha do arquivo principal da sua aplicação, para que ele carregue as variáveis de ambiente antes de tudo:



```
require('dotenv').config()
```

E *voilà!* Está pronto.

A única diferença entre `dotenv` e `dotenv-safe` (que eu prefiro, aliás) é que o segundo exige um arquivo `.env.example` que descreva todas as variáveis de ambiente obrigatórias que sua aplicação necessita para funcionar (apenas nomes das variáveis, uma por linha). Caso na inicialização não seja encontrada alguma das variáveis, dá um erro e a aplicação não sobe.

...

Carregue suas dependências antecipadamente

Muitos desenvolvedores Node.js programam desta forma os seus `requires`:



```
app.get("/meu-servico", function(request, response) {  
  var datastore = require("db-module")(config);  
  datastore.get(req.query.someKey)  
  // etc, ...  
});
```

O problema com o código acima é que quando alguém faz uma requisição para `/meu-servico`, o código então irá carregar todos os arquivos requeridos por `"db-module"` - o que pode causar uma exceção caso estejam bugados. Adicionalmente, quando passamos `config` para o objeto pode acontecer um erro de configuração (ou até de conexão) que derrube o processo inteiro. Além disso, não sabemos quanto tempo aquele `require`, que funciona de maneira síncrona,

irá levar e isso é terrível pois ele bloqueia todas as requisições até que termine.

O problema só não é pior porque depois que o require sobre aquele módulo for chamado uma vez, ele ficará em cache. Sendo assim, todas as suas dependências devem ser carregadas e configuradas antecipadamente, o que ajudará a descobrir erros de conexão e configuração bem cedo e deixar o funcionamento da aplicação mais fluida.



Centralize a criação de conexões ao banco

Passe as conexões criadas para os subsistemas ao invés de ficar passando informações de conexão por parâmetro e permitindo que eles criem suas próprias conexões.

Claro que toda regra tem sua exceção caso venha a utilizar Node.js com bancos relacionais em PaaS, talvez tenha que abrir e fechar as conexões por conta própria e não poderá ficar repassando o objeto entre as funções, ainda assim você pode centralizar o acesso à dados em um módulo específico para tal, como um db.js, por exemplo.



Mindset funcional

Você já deve ter ouvido falar que funções são objetos de primeira classe em JavaScript, certo? Isto é dito porque em JS funções não são tratadas de maneira muito distinta de outros objetos, diferente de outras linguagens que costumam deixar muito clara a diferença entre funções e variáveis.

Programar com um mindset funcional é aproveitar essa característica inerente da linguagem e usar e abusar de funções coesas, desacopladas, que aumentem o reuso de código, a legibilidade do código, facilitem a criação de testes (que veremos mais pra frente) e muito mais. Algumas regras básicas que você pode adotar com JS “funcional” sem medo de errar são:

SRP - Single Responsibility Principle

Cada função deve ter uma única razão de existir e, portanto, uma única razão para ser alterada. Jamais crie funções que atuem como “canivetes suíços”. Isso garante uma alta coesão e é uma boa dica para criação de módulos em JS também, evitando módulos como “utils.js”.

DRY - Don't Repeat Yourself

Trechos de código com lógica parecida? Encapsule em uma função. Funções chamadas repetidas vezes? Encapsule em um laço. Funções repetidas entre projetos? Encapsule em um módulo. Etc.

Imutabilidade

Sempre que possível, trabalhe com objetos imutáveis. Isso se tornou bem mais fácil com a adição do 'const' no ES6.

Baixo Acoplamento

Lembra da definição matemática do que é uma função? Funções são relações de um conjunto A com um conjunto B, ou seja, dados determinados parâmetros de entrada (A), teremos determinado retorno como saída (B). Sempre relacione A com B, jamais A com C (variáveis externas?) ou C com B, ou seja, para manter um baixo acoplamento a função apenas deve conhecer o conjunto de entradas para construir a partir dele as suas saídas.

Declarativo ao invés de Imperativo

Use as funções declarativas nativas como `foreach`, `map`, `reduce`, `filter`, etc trabalhem duro por você sem se preocupar com os detalhes de tais lógicas fica mais legível e muitas vezes mais eficiente.

Anonymous Functions

Aprenda a usar e se acostume com elas, especialmente closures e arrow functions.

Desde o ES6 que JS tem obtido características mais orientadas à objetos e alguns conceitos de OO inclusive foram citados logo acima. No entanto, focar em um JS OO, na minha opinião, não agrega tanto valor à linguagem quanto suas características funcionais.

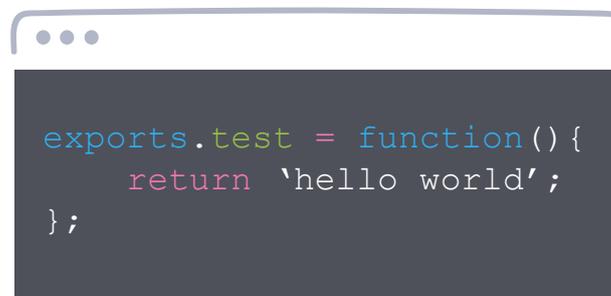
Existem bons livros sobre programação funcional, inclusive com JavaScript, além de cursos específicos, então não entrarei em detalhes aqui. Apenas reservo este espaço para alertar sobre a importância desse mindset para o uso correto desta tecnologia e sugerir a leitura deste material simples, porém muito inteligente: [JAMES SINCLAIR](#).

JS Full-stack

Muitas pessoas chegam até o Node com a promessa de escrever tanto o client-side quanto o server-side na mesma linguagem. No entanto, para que realmente isso seja vantajoso o reuso de software tem que ser possível em ambos os lados da aplicação, certo?!

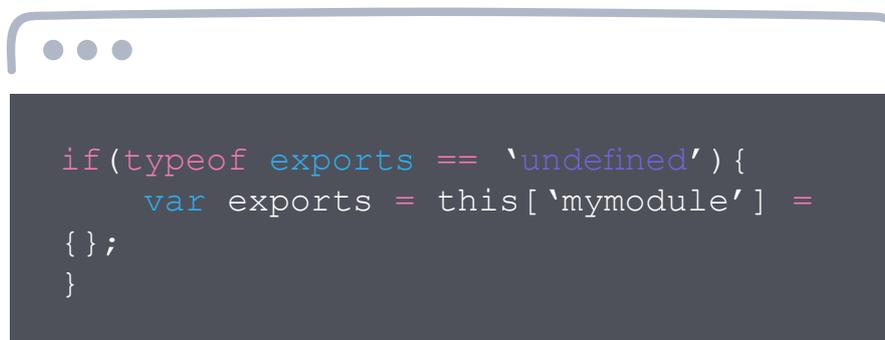
Uma boa prática é conseguir reutilizar os seus módulos JS tanto no browser quanto no back-end (via Node.js).

Primeiramente, em Node quando queremos expor um módulo, usamos um



```
exports.test = function() {  
  return 'hello world';  
};
```

No entanto, no browser isso dá erro uma vez que exports é undefined neste ambiente. O primeiro passo para contornar este problema é verificar a existência ou não do exports, caso contrário, teremos que criar um objeto para o qual possamos exportar as funções:



```
if(typeof exports == 'undefined') {  
  var exports = this['mymodule'] =  
  {};  
}
```

código semelhante a esse:

O problema com essa abordagem é que no browser as funções que não foram exportadas também ficam disponíveis como funções globais, o que não é algo desejável. É possível resolver isso usando closures, como no exemplo abaixo:

```
(function(exports) {

    // seu código vai aqui
    exports.test = function() {
        return 'hello world'
    };

})(typeof exports === 'undefined' ?
this['mymodule']={}: exports);
```

O uso do objeto `this` representa o browser, e o uso de `this['mymodule']` é o local de exportação no browser. Esse código está pronto para ser usado tanto no browser quanto no server. Considerando que ele está em um arquivo `mymodule.js`, usamos esse módulo da seguinte maneira em Node:

```
var mymodule = require('./mymodule'),
    sys = require('sys');

sys.puts(mymodule.test());
```

E no browser usamos assim:

```
<script src="mymodule.js"></script>
<script>
  alert(mymodule.test());
</script>
```

Claro, existem códigos JS escritos em Node que não serão suportados pelo browser, como o comando `require` por exemplo. Sendo assim, os módulos compartilhados entre os dois deverão ser o mais genéricos possíveis para que haja compatibilidade.

Tome cuidado também com as features mais recentes da linguagem Javascript que muitas vezes são suportadas pela engine V8 que o Node usa mas não pelos demais browsers e vice-versa (em casos mais raros). Nestes casos é muito útil usar um transpiler como o [BABEL](#).



TESTANDO



Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian W. Kernighan



Parece um tanto óbvio e estúpido incluir um capítulo sobre testes dentro de um guia de boas práticas, certo?

Quem dera fosse.

A grande maioria dos projetos que vivenciei (e ouvi falar), independente da plataforma, não possuem uma boa cobertura de testes, não fazem testes unitários corretamente, não possuem rotinas de regressão, etc. E nem estou falando do quase utópico TDD.

Coloque em sua mente que não importa a “idade” do seu projeto, nunca é tarde para iniciar uma cultura de testes. E não, testes não prejudicam projetos. Bugs sim. Cliente insatisfeito sim. Comece pequeno, comece simples, no mínimo com as dicas que serão dadas neste capítulo.



Use uma boa ferramenta

Assim como o JavaScript tradicional, em Node podemos usar qualquer editor de texto para programar nossas aplicações. No entanto, opções simplificadas demais como o Notepad e editores de linha de comando (como nano), embora práticos de usar, não são muito úteis quando precisamos depurar aplicações “bugadas”. Além disso, não contam com de outros benefícios como navegar entre funções, diversos tipos de highlights e autocompletes, etc.

Vou dar duas sugestões de como depurar programas Node.js.

Opção 1: troque seu editor de texto

O melhor jeito para mim atualmente é usando o [Visual Studio Code](#), que vem com suporte nativo a Node.js, é gratuito e roda em Windows, Mac e Linux. Nele você pode colocar breakpoints em arquivos .js, inspecionar variáveis, testar expressões no console, fazer integração com Git, [StandardJS](#) e muito mais.

Outra alternativa, mais pesada é o [Visual Studio Community](#) (2015 com [plugin para Node](#) ou 2017 com suporte nativo).

Opção 2: use um depurador externo ao seu editor

Agora se você não quiser abrir mão de usar editores de texto nativos do seu SO ou os clássicos com os quais já está acostumado, você pode depurar seus programas Node.js diretamente no Google Chrome também, usando o F12. Você consegue mais informações neste [post do Medium](#).

E, por fim, uma outra alternativa para não largar seus editores de texto favoritos é usando o Node Inspector, um projeto [open-source disponível no GitHub](#).



Trate bem os seus erros

Não há nada mais agradável do que de repente a sua aplicação Node sair do ar e você descobrir no servidor que foi uma exception que fez isso. Aquele trecho de código mal testado, aquela resposta não prevista da API, etc. Não importa o que era, mas sim que derrubou tudo. E você não pode deixar isso acontecer.

Um bom gerenciamento de exceções é importante para qualquer aplicação, e a melhor maneira para lidar com erros é usar as ferramentas fornecidas pelo Node.js para isso, como em promises, em que temos o handler `.catch()`, que propaga todos os erros para serem tratados em um único local, como no exemplo abaixo:



```
doSomething()
  .then(doNextStage)
  .then(recordTheWorkSoFar)
  .then(updateAnyInterestedParties)
  .then(tidyUp)
  .catch(errorHandler);
```

Qualquer erro que aconteça nos `then()` será capturado para tratamento no `catch()`.

Aumente a visibilidade dos erros

A dica aqui não é jogar erro para o usuário, mas sim usar uma biblioteca de logging para aumentar a visibilidade dos erros. O `console.log` é ótimo, mas tem sérias limitações em uma aplicação em produção. Tentar encontrar a causa de um bug dentre milhares de linhas de logs é terrível e uma biblioteca de logging madura pode ajudar com isso.

Primeiro, bibliotecas de logging permitem definir níveis para as mensagens como `debug`, `info`, `warning` ou `error` mesmo. Segundo, geralmente elas permitem quebrar os logs em diferentes arquivos ou mesmo persistir remotamente, o que, aliás, é uma ótima ideia na minha opinião.

Uma sugestão muito boa é mandar seus logs para a [Loggly.com](https://loggly.com) que tem um plano free bem interessante de 200MB de logs por dia e armazenamento dos últimos sete dias. O serviço dessa startup permite buscar rapidamente por mensagens usando padrões e definir alertas caso muitos erros estejam acontecendo.

Falando de bibliotecas especificamente, a recomendada aqui é a [winston](#).



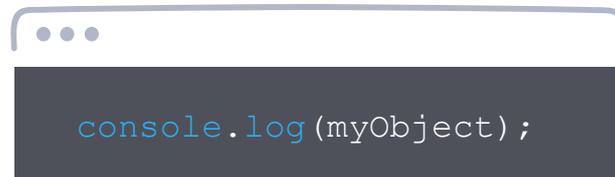
Vendo os objetos completamente no console

Certas vezes quando temos objetos complexos em Node.js e queremos ver o que ele está guardando dentro de si usamos o console do Google Chrome ou mesmo do Visual Studio para entender o que se passa com nosso objeto. No entanto, dependendo do quão “profundo” é o nosso objeto (quantos objetos ele possui dentro de si), essa tarefa não é muito fácil.

Aqui vão algumas formas de imprimir no console o seu objeto JSON inteiro, não importando quantos níveis hierárquicos ele tenha:

Opção 1: console.log

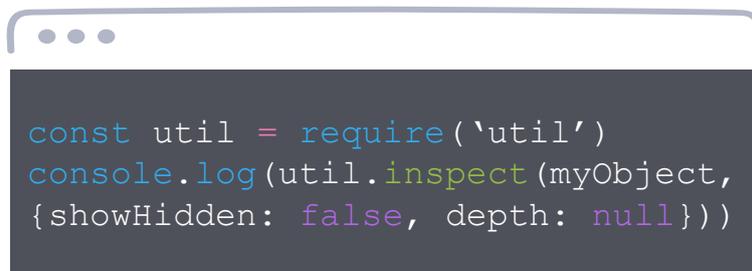
Tente usar a função console.log passando o objeto or parâmetro, isso funciona na maioria dos casos.



```
console.log(myObject);
```

Opção 2: util.inspect

Use o seguinte código abaixo para usar a função util.inspect e retornar todo o conteúdo de um objeto JSON.



```
const util = require('util')
console.log(util.inspect(myObject,
{showHidden: false, depth: null}))
```

Opção 3: JSON.stringify

Use a função JSON.stringify passando o objeto e o nível de indentação que deseja dentro do objeto, como abaixo.



```
console.log(JSON.stringify(myObject, null, 4));
```

Escreva testes unitários

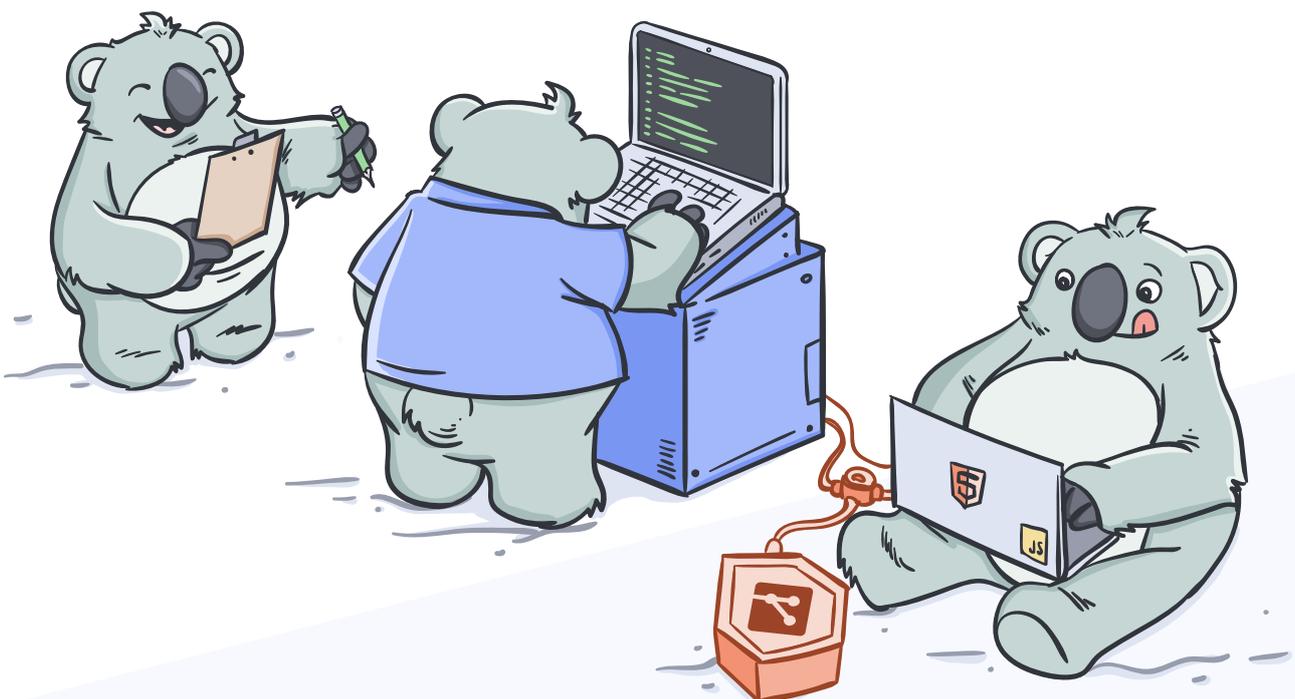
Escrever testes unitários permite que você tenha mais confiança que as menores partes do seu projeto funcionam como deveriam. Além disso, resolve a maior parte dos bugs toscos que tiram muitas noites de sono dos desenvolvedores. Unit tests são facilmente automatizados através de extensões adequadas, oferecendo um ganho de qualidade e velocidade muito grande.

Existem diversos módulos para ajudar com testes unitários em Node.js e não há um consenso sobre qual o melhor deles, sendo os mais comumente recomendados:

- Jasmine
- Mocha
- Chai
- Tap

Especificamente para web applications e web APIs, usar o [Supertest](#) ou o [hippie](#) para abstrair a camada de front-end é uma ótima ideia também.

Não vou entrar em detalhes aqui, já que não sei qual biblioteca você vai adotar e todas possuem excelente documentação e tutoriais na Internet. Os excelentes livros do Kent Beck como TDD (ele é o inventor do termo) e até mesmo o livro Programação Extrema (seu método ágil de desenvolvimento de software) são altamente recomendados neste tópico.



DEPLOY



The most important property of a program is whether it accomplishes the intention of its user.

C.A.R. Hoare



Então você usou os melhores princípios do Node.js, escreveu sua aplicação perfeitamente, depurou e tratou todos os erros, agora é só colocar em produção, certo?

‘Só’ é uma palavra meio fraca para esta tarefa que, dependendo da sua aplicação, pode exigir bastante trabalho ou ao menos bastante configuração de sua parte. Há ainda a questão de que apenas ‘colocar em produção’ não resolve todo seu problema, afinal você terá de ter mecanismos que mantenham a sua aplicação disponível, mecanismos de logging, etc.

Neste capítulo falaremos de várias coisas sobre colocar sua aplicação no ar e mantê-la lá!



Não use FTP!

Antes de entrar nas boas práticas do que você deve usar, falarei do que você não deve usar: FTP. Estamos falando de uma tecnologia da década de 80 que, embora ainda seja utilizada até hoje em diversos contextos, nunca foi criada com o objetivo de implantar software web.

Implantar software na web profissionalmente raramente é apenas subir arquivos para uma pasta no seu servidor. Geralmente envolve fazer um backup da versão atual em produção, subir a nova versão, substituir as duas, etc. No caso do Node.js, pode envolver também baixar os módulos corretos via NPM e esse é o tipo de coisa que você não vai querer subir via FTP.

Se você usa um provedor de hospedagem para manter suas aplicações Node.js 24x7, tenha em mente que FTP inevitavelmente falha e que isso não é bom para deploy em produção. Afinal, quem nunca ficou com um arquivo trancado em 0 bytes no servidor e você não consegue nem excluí-lo, bem sobresscrevê-lo?

Na Umblar nossa hospedagem Node.js não usa FTP, mas sim Git para deploy. Na nossa abordagem, tanto faz se você enviar a sua pasta `node_modules` ou não, e você pode também adicionar outros usuários com permissão para fazer deploy também, se necessário.

Cuide da segurança

Se o seu projeto é uma web application, existem um monte de boas práticas de segurança específicas visando manter sua app segura:

- Proteção a XSS;
- Prevenção de ClickingJacking usando X-Frame-Options;
- Forçar conexões como HTTPS;
- Configurar um cabeçalho Context-Security-Policy;
- Desabilitar o cabeçalho X-Powered-By para que os atacantes não tenham informações úteis para ataques;

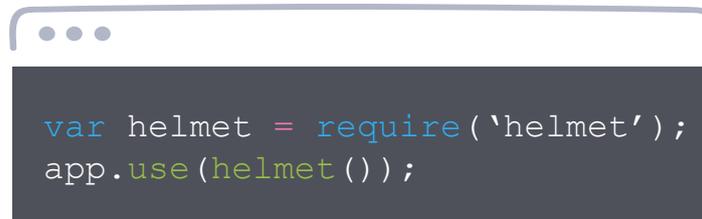
Ao invés de tentar se lembrar de todas elas você pode usar o módulo Helmet, que permite facilmente alterar todas configurações default (e inseguras) de web applications e deixa você customizar as que forem necessárias.

Como tudo em Node.js, é incrivelmente simples de instalar o Helmet:



```
$ npm install helmet
```

e muito simples e usar na sua aplicação:



```
var helmet = require('helmet');  
app.use(helmet());
```

...

Mantendo seu processo no ar!

Se você optar por uma solução hands-on, em que você terá de configurar o servidor para manter sua aplicação rodando, certifique-se de fazer com que seu processo fique sempre rodando, mesmo que seja derrubado e mesmo que o servidor inteiro reinicie. Não importa se você tratou todos os erros da sua aplicação, inevitavelmente algo vai acontecer e tirar seu processo do ar e ele deve ser capaz de subir novamente.

Aqui vão algumas boas opções de como fazer isso:

Windows Service

Uma das opções é instalar um script Node.js de inicialização da sua aplicação como um Windows Service (algo bem parecido com o que você deve ter feito na dica de codificação sobre scripts no package.json). Um Windows Service permite configurações como iniciar junto ao Windows (caso ele reinicie), reiniciar automaticamente em caso do processo travar, os erros enfrentados pelo Windows Service vão parar no Event Viewer do Windows, etc.

Para fazer isso, primeiro instale o [módulo node-windows](#) globalmente:

```
npm install -g node-windows
```

Agora rode o seguinte comando (dentro da pasta do seu projeto) para incluir uma referência deste módulo ao seu projeto:

```
npm link node-windows
```

Depois, dentro do seu projeto Node.js (na raiz mesmo) crie um arquivo service.js com o seguinte conteúdo:

```
var Service = require('node-windows').Service;

// Create a new service object
var svc = new Service({ name: 'Nome da sua Aplicação',
description: 'Apenas uma descrição', script:
'C:\\domains\\sitenodejs\\bin\\www' });

// Listen for the "install" event, which indicates the
// process is available as a service.
svc.on('install', function() { svc.start(); });
svc.install();
```

Troque as propriedades name e description de acordo com seu gosto, mas atente à propriedade script nesse código, que deve conter o caminho absoluto até o arquivo JS que inicia sua aplicação. Neste exemplo estava usando ExpressJS e aponte para o arquivo www que fica na pasta bin do projeto (curiosamente ele não possui extensão, mas é um arquivo).

Se você fez tudo corretamente, vá até Ferramentas Administrativas > Serviços (Administrative Tools > Services ou services.msc no Run/Executar do Windows) e seu serviço vai aparecer lá com o nome que definiu ali no script, permitindo que você altere suas configurações de inicialização, dar Start, Stop, etc.



```
SC STOP servicename
SC DELETE servicename
```

Caso precise remover esse serviço (para instalar uma versão mais atualizada, por exemplo) rode o comando abaixo no cmd:

Esse servicename você encontra nas propriedades do Windows Service que deseja excluir.

PM2

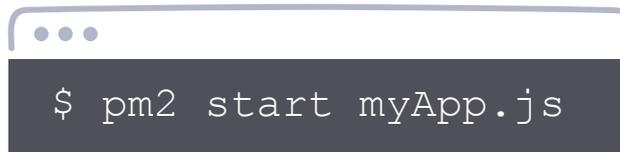
PM2 é um projeto open-source criado e mantido pela empresa Keymetrics. io, que, além do PM2 (que é gratuito), vende um serviço de gerenciamento de performance de aplicações Node.js homônimo. Só para você ter uma ideia do que o PM2 é hoje, são mais de 20M de downloads e empresas como IBM, Microsoft e PayPal usando, o que o torna, disparado, a melhor solução de process manager pra Node, muito mais do que seus principais concorrentes, o Forever e o Nodemon.

Para usar o PM2 é muito simples, primeiro instale globalmente o módulo do PM2:



```
$ npm install pm2 -g
```

Depois, quando quiser iniciar o processo da sua aplicação Node.js:



```
$ pm2 start myApp.js
```

Para garantir que ele rodará eternamente, se recuperará sozinho de falhas, etc, você pode estudar o [guia oficial](#) dele.

Além do PM2, a Keymetrics.io oferece uma ferramenta de monitoramento e alertas muito boa e útil que inclusive conta com um plano gratuito que atende boa parte dos desenvolvedores. Usar uma boa ferramenta de monitoramento ajuda bastante a evitar dores de cabeça principalmente se seu serviço possui clientes pagantes. Ao contrário de ferramentas de monitoramento de uptime simples como Pingdom e StatusCake, a Keymetrics.io monitora consumo de recursos como RAM e CPU.



Trabalhando com vários cores

O Node.js trabalha com uma única thread dentro de um único processo na sua máquina. Dessa forma, é natural que ele utilize apenas um processador, mesmo que você esteja rodando sua aplicação em um webserver com 16 núcleos ou mais. Sendo assim, uma dúvida bem comum é: como escalar um projeto Node para que use todo o poder do seu servidor?

Basicamente você tem algumas opções de como fazer o deploy de uma aplicação Node.js que use os diversos núcleos da máquina:

Usar uma arquitetura de micro-serviços.

Cada módulo da sua aplicação deve ser uma sub-aplicação autônoma, que responde a requisições e realiza apenas as tarefas que são de sua responsabilidade. Sendo assim, teríamos diversas aplicações pequenas escritas em Node.js, cada uma usando um core da sua máquina, recebendo (e processando) as requisições que cabem a elas. Uma aplicação principal recebe a requisição original do usuário e delega as tarefas para as demais sub-aplicações. Falei disso em mais detalhes no capítulo de Codificação.

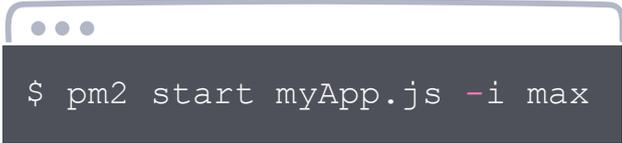
Note que nesta abordagem usaremos vários cores pois teremos vários serviços Node rodando e, conseqüentemente, vários processos. Dentro da plataforma da Umblar você consegue adotar esta estratégia com vários subdomínios apontando para instâncias pequenas ao invés de uma instância grande rodando um único app Node.js.

Usar um “webproxy” na frente do Node

Você pode colocar um Apache, Nginx ou IIS à frente da sua aplicação e deixar com ele essa tarefa de controlar a carga de requisições, balanceando entre diferentes nós idênticos da sua aplicação, cada um em um processador. Você pode fazer isso com Node.js também, mas geralmente Apache e cia. já possuem muito mais maturidade para isso.

Nesta abordagem usaremos vários cores rodando o mesmo serviço de maneira repetida. Tenha isto em mente quando estiver criando sua aplicação, pois cada processo irá fazer a sua conexão com o banco e não compartilhará memória com os demais. Uma alternativa para solucionar esse problema de memória compartilhada é usar um Redis ou similar para compartilhar recursos.

Além disso, ao invés de usar um webproxy à frente do Node você pode usar o PM2. Basicamente você consegue subir uma instância da sua aplicação Node por CPU usando o comando abaixo:



```
$ pm2 start myApp.js -i max
```

Consulte o [guia oficial do PM2](#) para conhecer mais sobre clusterização com ele. Existem outras opções? Sim, mas essas duas acima são recomendadas.

Use Integração/entrega/implantação contínua

Algumas definições rápidas pois existe muita confusão sobre estes termos:

Continuous Integration

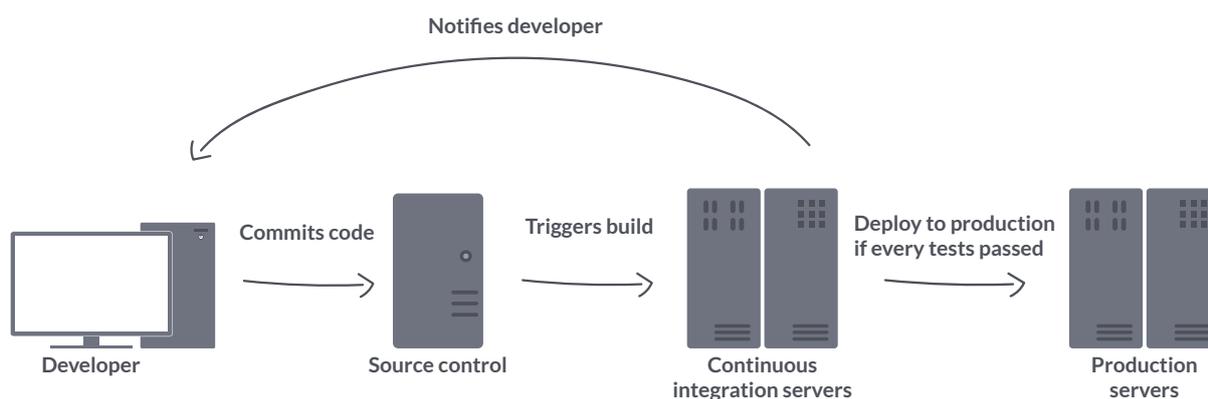
É o processo de fundir o trabalho desenvolvido com a master várias vezes ao dia e/ou constantemente (dependendo do seu ritmo de trabalho), ajudando a capturar falhas rapidamente e evitar problemas de integração, sendo boa parte desses objetivos atingidos com uma boa bateria de testes automatizados.

Continuous Delivery

É o próximo passo após a entrega contínua, sendo a prática de entregar código para um ambiente, seja para o time de QA ou para clientes, para que o mesmo seja revisado. Uma vez que as alterações sejam aprovadas, eles podem ir para produção.

Continuous Deployment

O último passo nessa sequência é a implantação contínua, quando o código que passou pelas etapas anteriores com sucesso é implantado em produção automaticamente. Continuous deployment depende pesadamente de uma infraestrutura organizada e automatizada.



Fonte: Rising Stack

Embora estas definições variem um pouco dependendo da linha de pensamento adotada pelos autores de livros e professores universitários, todas elas estão associadas a conceitos mais universais como ALM e Agile e até mesmo com necessidades bem específicas como as entregas contínuas do Scrum e do Lean Startup. Independente disso, têm se mostrado cada vez mais importantes, e valiosas, nos projetos atuais.

Felizmente, Node.js parece ter nascido para ser entregue continuamente pois possui diversas características (cortesia do JavaScript) propícias para tal como a modularização, a dinamicidade, a facilidade de uso como micro serviços, entre muitas outras.

Considero que existem cinco pilares a se terem em mente se o objetivo é alcançar um ambiente de entrega contínua realmente funcional:

Pilar 1: Versionamento

Seu código deve estar versionado em um repositório (geralmente Git), pois todo o processo de integração começa com um commit.

Pilar 2: Testes automatizados

O commit deve disparar (ou ao menos ser precedido, você escolhe) por uma bateria de testes automatizados, o que inclusive pode ser feito com as bibliotecas citadas no capítulo sobre testes.

Uma boa cobertura de testes é crucial se você quiser ser bem sucedido com entrega contínua, caso contrário a estratégia pode sair pela culatra, com bugs sendo entregues em produção mais rápido do que nunca!

Pilar 3: Chaveamento de features

Como os commits vão parar na master bem cedo, geralmente antes do final do projeto, é natural que existam features que devem esperar para serem lançadas e outras que devem ser lançadas imediatamente. O chaveamento de features (feature toggle no original) também é muito útil quando se deseja subir correções pra master sem adicionar as funcionalidades novas nas quais você ainda está trabalhando e são uma alternativa para as dores de cabeça com branches. Existe inclusive um módulo no NPM chamado feature-toggles com esse objetivo.

Pilar 4: Ferramenta de CI

Existem diversas opções no mercado, tanto open-source quanto proprietárias, tanto pagas quanto gratuitas, fica a seu critério. A dica aqui é: não reinvente a roda. Existem excelentes opções consolidadas e bem testadas, apenas escolha uma e siga em frente.

- Algumas opções são:
- Jenkins (open-source)
- Travis
- Codeship
- Strider (open-source)
- CircleCI

Basicamente o que essas ferramentas fazem são orquestrar builds complexos que são iniciados a partir de commits no seu repositório em uma branch específica. Esses builds podem incluir testes automatizados, execução de scripts, transferência de arquivos, notificações por email e muito mais. Tudo vai depender do que seu projeto precisa.

Pilar 5: Rollback

Sempre tenha em mente que o processo pode falhar, ainda mais em uma arquitetura complexa de deployment contínuo e que, se isso acontecer, tudo deve continuar como estava antes do processo de CI ter sido inicializado. Um rollback bem pensado e 100% funcional deve ser uma prioridade.

Com certeza não estressei esse assunto com estas poucas sugestões. Existem excelentes livros sobre ALM, CI, etc mas espero ter despertado no mínimo o interesse pelo assunto caso seja novidade para você. Na [Umblor](#) facilitamos esses processos através do deploy via Git e essa é uma das características mais apreciadas por nossos clientes.



MÓDULOS RECOMENDADOS

Simple things should be simple,
complex things should be possible.

Alan Kay



Uma das coisas mais fantásticas do Node.js é o seu ecossistema. O NPM rapidamente cresceu para se tornar o maior repositório de bibliotecas e extensões do mundo e isso em menos de uma década de existência.

Esse capítulo, na verdade, é um índice remissivo para que você encontre facilmente todos os módulos recomendados nos demais capítulos, apenas por uma questão de organização, em ordem alfabética e com uma frase resumindo do que se trata. Tomei a liberdade de incluir também outras bibliotecas notáveis que não foram citadas, mas que acredito serem importantes.

Clique nos links dos nomes para ir ao site de cada módulo.

Artillery.io

Módulo muito bacana para testes de carga e stress em APIs, não necessariamente Node.

Babel

Transpilador muito utilizado para permitir a utilização de recursos recentes do EcmaScript mesmo em navegadores mais antigos. Mencionado no tópico sobre JS Full-Stack no Capítulo 3.

Chai

Popular biblioteca para uso de TDD e BDD com Node.js. Mencionado no tópico sobre testes unitários no capítulo 4.

DotEnv e DotEnv-Safe

Módulos para carregamento de variáveis de ambiente de maneira rápida e fácil. A diferença entre os dois pacotes é que a versão 'safe' não permite a inicialização do sistema se as variáveis de ambiente necessárias não puderam ser encontradas. Citados no tópico de variáveis de ambiente do capítulo 3.

EJS (Embedded JavaScript)

View-engine muito popular para usa com Express visando permitir o uso de HTML e JS para composição das views.

Express

Web framework extremamente popular para Node.js. Se você não conhece, deveria, resolve boa parte das demandas com Node de maneira rápida e fácil.

Express Generator

Um scaffold para ExpressJS que já define uma arquitetura básica mas bem útil para iniciar rapidamente com projetos Node.js.

Feature-toggles

Extensão para permitir facilmente fazer...feature-toggles. Citado no tópico sobre integração contínua no capítulo 5.

Helmet

Módulo de segurança que já blinda a sua aplicação Express quanto aos ataques e brechas mais comuns em web applications. Citado no tópico de segurança no capítulo 5.

Hippie

Extensão para permitir o teste de APIs web abstraíndo boa parte das complicações. Mencionado no tópico sobre testes unitários no capítulo 4.

Jasmine

Popular biblioteca para testes. Mencionado no tópico sobre testes unitários no capítulo 4.

Loopback.io

Framework web muito popular para criação de APIs Restful.

Mocha

Popular biblioteca para testes. Mencionado no tópico sobre testes unitários no capítulo 4.

Mongoose

ORM bem popular e poderoso, especialmente para uso com MongoDB, até 50% mais veloz do que seu principal concorrente, Monk.

Node-Windows

Biblioteca para uso de scripts Node.js como Windows Service em servidores. Citado no tópico de como manter seu processo no ar no capítulo 5.

Passport

Autenticação em aplicações Node.js, especialmente APIs.

PM2

Gerenciador de processos para Node.js, visando recuperação de falhas, monitoramento e muito mais. Citado no tópico de como manter seu processo no ar no capítulo 5.

Sequelize

ORM muito popular e indicado para uso de Node.js com bancos relacionais.

Socket.io

Comunicação ponto-a-ponto para aplicações de mensageria em Node.js.

StandardJS

Linter JavaScript que determina um guia de estilo uniforme e popular para codificação de projetos JS. Citado no tópico de guia de estilo no capítulo 2.

Supertest

Extensão para permitir o teste de aplicações web abstraindo o front-end. Mencionado no tópico sobre testes unitários no capítulo 4.

Tap

Popular biblioteca para testes. Mencionado no tópico sobre testes unitários no capítulo 4.

Typescript

Um superset que permite escrever em um JavaScript mais 'bombado' que depois é compilado para JavaScript tradicional para uso em produção.

Winston

Popular (e poderosa) biblioteca para logging. Citado no tópico sobre visibilidade dos erros no capítulo 4.

Tem algum módulo que você conhece e recomenda e que não está nessa lista? Mande para nós pelo amigos@umbl.com que incluiremos em revisões futuras deste guia!

SEGUINDO EM FRENTE



A code is like love, it has created with clear intentions at the beginning, but it can get complicated.

Gerry Geek



Este guia termina aqui.

Pois é, certamente você está agora com uma vontade louca de aprender mais e criar aplicações incríveis com Node.js, que resolvam problemas das empresas e de quebra que o deixem cheio de dinheiro na conta bancária, não é mesmo?

Pois é, eu também! :)

Este guia é propositalmente pequeno, com menos de 40 páginas. Como professor, costumo dividir o aprendizado de alguma tecnologia (como Node.js) em duas grandes etapas: aprender o básico e executar o que foi aprendido no mercado, para alcançar os níveis intermediários e avançados. Acho que este guia atende bem ao primeiro requisito, mas o segundo só depende de você.

De nada adianta saber muita teoria se você não aplicar ela. Então agora que terminou de ler este guia e já conhece uma série de boas práticas com esta fantástica plataforma, inicie hoje mesmo (não importa se for tarde) um projeto de aplicação que as use. Caso não tenha nenhuma ideia, cadastre-se agora mesmo em alguma plataforma de freelancing. Mesmo que não ganhe muito dinheiro em seus primeiros projetos, somente chegarão os projetos grandes, bem pagos e realmente interessantes depois que você tiver experiência.

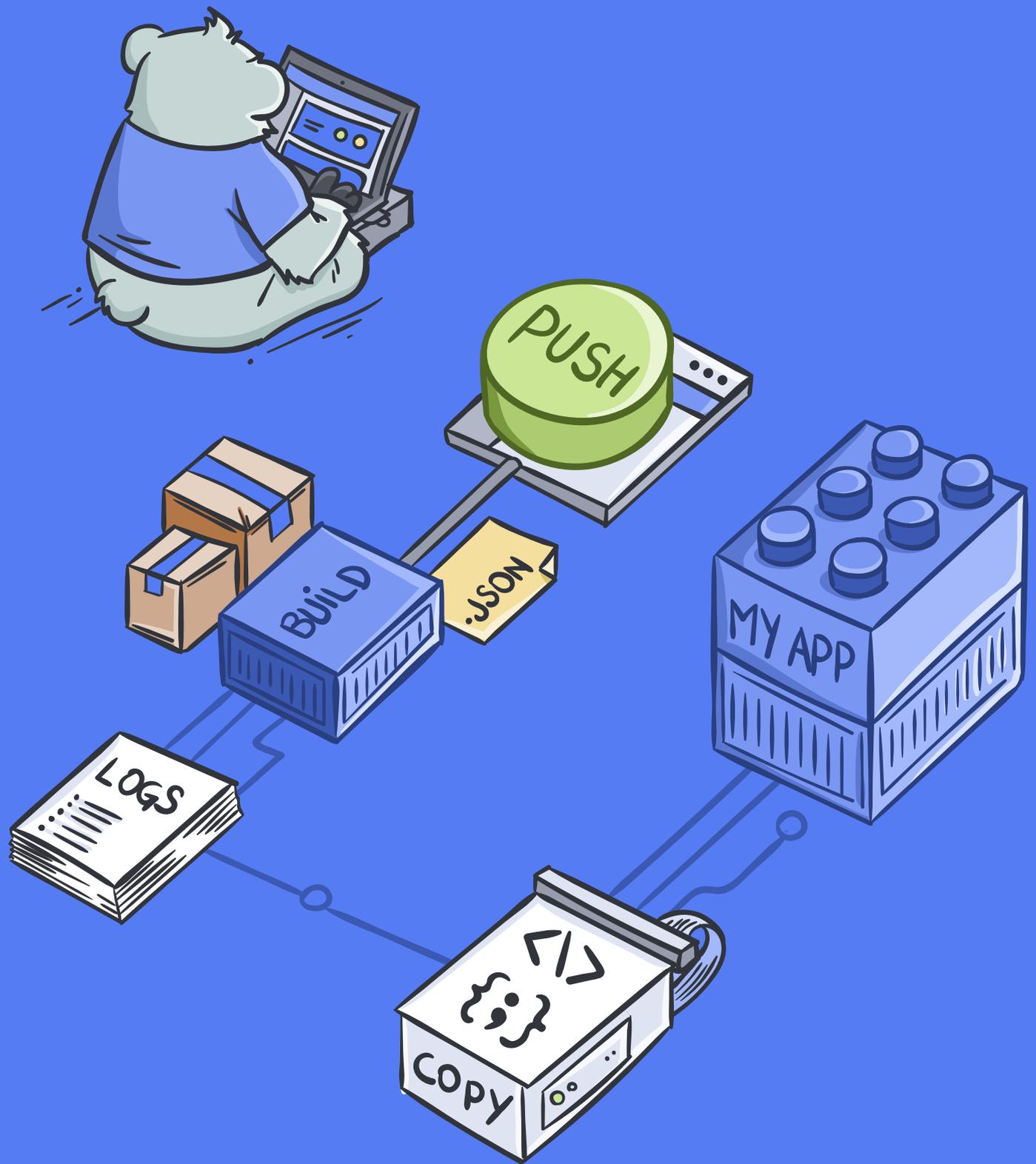
Me despeço de você leitor com uma sensação de dever cumprido. Caso acredite que está pronto para conceitos mais avançados, sugiro dar uma olhada em meu blog [LuizTools](#) e na página [Academy](#).

Outras fontes excelentes de conhecimentos específico de Node é o blog [RisingStack](#), enquanto que sobre JavaScript a [Developer](#) possui tudo que você pode precisar!

Caso tenha gostado do material, indique esse ebook a um amigo que também deseja aperfeiçoar as suas habilidades com Node. Não tenha medo da concorrência e abraça a ideia de ter um sócio que possa lhe ajudar nos projetos.

Caso não tenha gostado tanto assim, envie suas dúvidas, críticas e sugestões para amigos@umbler.com que estamos sempre dispostos a melhorar.

Um abraço e até a próxima!



umblar