Apostila NodeJS - 16 horas - CT Novatec

Esta é a apostila do curso presencial de NodeJS de 16 horas que irei ministrar no Centro de Treinamento Novatec (http://ctnovatec.com.br/cursos/trilha-front-end/curso-de-nodejs/).

Instrutor

Eu sou o **William Bruno**, aka **wbruno**, desenvolvedor web apaixonado por JavaScript. Você pode ler alguns artigos que escrevi no meu **blog pessoal** (http://wbruno.com.br), para o portal **iMasters**(http://imasters.com.br/perfil/william_bruno/), para o blog **UOL Host** e mais alguns outros lugares por ai. Caso você participe do **fórum imasters**, poderá me encontrar por lá também. (http://forum.imasters.com.br/user/69222-william-bruno/).

Outras formas de contato:
https://about.me/wbruno
<wbrunom@gmail.com>

O conteúdo

Eu gravei um Workshop que está publicado na **Eventials** sobre como **Construir uma API REST com ExpressJS** (https://www.eventials.com/wbruno.moraes/construindo-uma-api-rest-com-expressjs-nodejs-2/), que é exatamente uma parte do foco desse curso presencial.

Nessa apostila irei abordar com mais “calma” e mais detalhadamente cada etapa desde a construção da API até a listagem dos produtos no site e da edição no admin. Depois escreveremos alguns testes unitários e funcionais para garantir que o nosso código seja confiável e sólido.

Sempre que possível irei colocar links relacionados para que você consiga complementar seus estudos e use como guia de referência em dúvidas sobre o tema.

Desafios

O HackerRank é uma plataforma online com desafios de programação:
https://www.hackerrank.com/domains

Introdução

O que é o NodeJS?

NodeJS é uma plataforma escrita em cima da engine JavaScript do Chrome, a poderosa **V8**, para construir escaláveis, utilizamos a linguagem JavaScript. https://nodejs.org/

Primeiros passos
http://tableless.com.br/o-que-nodejs-primeiros-passos-com-node-js/

Instalando o NodeJS

Vou deixar alguns links sobre como instalar, basta seguir as instruções corespondentes ao seu sistema operacional.

- **Todas** http://nodebr.com/instalando-node-js-atraves-do-gerenciador-de-pacotes/
- **Mac OSx** http://udgwebdev.com/node-js-para-leigos-instalacao-e-configuracao/
- **Windows** http://mateussouzaweb.com/blog/nodejs/tutorial-instalando-nodejs-no-windows

O npm (Node Package Manager)

O que é
https://docs.npmjs.com/getting-started/what-is-npm

O npm (https://www.npmjs.com) é um serviço grátis (para pacotes públicos) que possibilita que os desenvolvedores NodeJS criem e compartihem códigos com a comunidade. Facilitando a distribuição de módulos e ferramentas escritas em JavaScript.

Por exemplo, se você quiser utilizar o **ExpressJS** (http://expressjs.com) que é um framework rápido e minimalista para construção de rotas em NodeJS, você não precisa entrar no github deles, baixar o código, e "instalar" (colar) no seu projeto. Basta digitar no seu terminal:

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

Que o módulo será baixado dentro de uma pasta chamada `node_modules` e pronto, você já pode utilizar o express.

O lado bom

O lado “bom” do npm é que existem muitos módulos a disposição. Então sempre que você tiver que fazer alguma coisa em NodeJS, vale a pena dar uma pesquisada no https://www.npmjs.com, se já existe algum módulo que faça o que você quer, ou uma parte do que você precisa. Agilizando muito assim o desenvolvimento da sua aplicação.

O lado ruim

O lado “ruim” do npm é que existem muitos módulos a disposição! Opa, mas eu não acabei de falar que esse era o lado bom ? pois é, acontece que por ser completamente público e colaborativo (*Open Source*), você encontrará diversos módulos com o mesmo propósito, que realizam as mesmas tarefas. Cabe a nós conseguir escolher aquele que melhor resolve a nossa questão, e para isso eu costumo seguir alguns simples passos:

- Procure um módulo que esteja sendo mantido (com atualizações frequentes, olhe se o último commit não foi a anos atrás,mas sim alguns dias ou semanas).
- Se existe alguma comunidade ativa utilizando o módulo (olhe o número de estrelas, forks, issues abertas e fechadas…)
- Importante que ele possua uma boa documentação dos métodos públicos e da forma de uso, assim você não terá que ler o código fonte do projeto para realizar uma tarefa simples. (ler o fonte pode ser interessante, quando você tiver um tempo para isso, ou precisar de alguma otimização mais baixo nível).
- Procure por benchmarks onde são comparados módulos alternativos, e decida por aquele com melhor performance.

Assim você foca na sua aplicação e em desenvolver os códigos da regra de negócio.

O arquivo package.json

Todo projeto NodeJS contém um arquivo chamado `package.json` na raiz. Esse arquivo contém informações sobre a aplicação, assim como as dependências dela.
Sempre use o comando `npm init` para criar a estrutura inicial do seu `package.json`.

```
$ npm init
```

Basta ir respondendo as perguntas, uma a uma, como “nome da aplicação”, “versão”, “link do repositório git”, no final, confirme com um `yes`, e um arquivo mais ou menos parecido com esse será criado:

```
{
  "name": "curso-nodejs-16h-novatec",
  "version": "0.0.1",
  "description": "Loja Virtual criada durante o curso de NodeJS
presencial no CT Novatec",
  "main": "app.js",
  "scripts": {
     "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "William Bruno <wbrunom@gmail.com>
(http://wbruno.com.br/)",
  "license": "ISC"
}
```

Feito isso, podemos começar.

GruntJS

Nem só para aplicações web vive o NodeJS. O GruntJS é um ótimo exemplo de como podemos utiliza-lo como uma ferramenta de linha de comando.
http://gruntjs.com

A idéia por trás do GruntJS é automatizar tarefas. Desde as mais simples como concatenar arquivos .css e .js, minificar o código (remover espaços, tabulações, encurtar nomes de variáveis) até fazer deploy!

Tudo depende de quais plugins você utilizar junto com ele.O GruntJS sozinho, apenas gerencia esses módulos e organiza a ordem e quais tarefas você irá utilizar, mas serão os módulos que você plugar que irão realizar os trabalhos, como por exemplo:

https://github.com/gruntjs/grunt-contrib-cssmin
https://github.com/gruntjs/grunt-contrib-uglify
https://github.com/gruntjs/grunt-contrib-watch

Um artigo complementar: http://simplesideias.com.br/usando-gruntjs
Você poderá encontrar muitos outros sobre esse assunto, caso sinta alguma dúvida ou precise implementar mais coisas no teu grunt.

Utilizando

Após instalarmos o NodeJS, em uma aba do terminal, precisamos do grunt-cli como um módulo global do computador, para podermos utilizar o comando 'grunt-cli', sem precisar identificar o diretório em que ele está instalado, e para qualquer uma das nossas aplicações ter acesso aos comandos.

```
$ npm install grunt-cli -g
```

Criamos o `package.json` desse projeto com o comando

```
$ npm init
```

E já podemos instalar as dependências locais, tais como:

```
$ npm install grunt grunt-contrib-watch grunt-contrib-uglify
grunt-contrib-cssmin --save-dev
```

Note que utilizei a flag `--save-dev` para que esses módulos entrem no package.json, mas na sessão "`devDependencies`", afinal não precisamos minificar css e javascript na hora de rodar a aplicação em produção, precisamos apenas em desenvolvimento, enquanto estivermos codando.

Todos esses módulos do grunt, ficarão dentro da pasta `node_modules` na raiz da nossa aplicação, no mesmo nível do arquivo `package.json`. Ficando assim lá:

```
"devDependencies": {
  "grunt": "^0.4.5",
  "grunt-contrib-cssmin": "^0.12.2",
  "grunt-contrib-uglify": "^0.9.1",
  "grunt-contrib-watch": "^0.6.1"
}
```

Gruntfile.js

O comando `grunt` irá sempre buscar um arquivo chamado `Gruntfile.js` na raiz do projeto. É nele onde iremos declarar as nossas tasks: concatenar e minificar css e js.

A estrutura inicial desse arquivo é:

```
module.exports = function (grunt) {
  'use strict';

  grunt.initConfig({
```

```
    pkg: grunt.file.readJSON('package.json')
  });

  grunt.registerTask('default', []);
};
```

Utilizando o CSS min

Basta adicionar essa propriedade no json do grunt:

```
cssmin: {
  with_banner: {
    options: {
      banner: '/*\n' +
           'Minified CSS of <%= pkg.name %>\n' +
           '*/\n'
    },
    files: {
      'public/css/all.min.css': [
           'src/css/normaset.css',
           'src/css/main.css'
      ]
    }
  }
},
```

O nosso objetivo é pegar os arquivos .css da pasta `src/css` e minifica-los em `public/css`

Utilizando o Uglify

Segue a mesma idéia:

```
uglify: {
  options: {
     banner: '/* Minified JavaScript of <%= pkg.name %> */\n'
  },
  my_target: {
    files: {
      'public/javascript/all.min.js': [
        'src/js/vendor/*.js',
        'src/js/*.js'
```

```
      ]
    }
  }
},
```

Utilizando o grunt watch

E por fim, vamos implementar o watch, para deixarmos o comando:

```
$ grunt watch
```

Rodando em uma aba no terminal, e assim que algum arquivo fonte `.css` ou `.js` for modificado, o grunt irá minificá-lo para nós.

```
watch: {
  options: {
     livereload: true,
     spawn: false
  },
  css: {
     files: 'src/css/*.css',
     tasks: ['cssmin']
  },
  js: {
     files: 'src/js/*.js',
     tasks: ['uglify']
  }
}
```

Além disso, adicionaremos o suporte ao livereload no nosso html, apenas incluindo essa tag script no html:

```
<script src="//localhost:35729/livereload.js"></script>
```

Assim, sempre que o grunt terminar alguma tarefa do watch, a nossa página será automaticamente recarregada, e não precisaremos quebrar a tecla F5 de tanto usá-la.

Nosso código até agora

https://github.com/wbruno/curso-nodejs-16h-novatec/tree/v0.1.0

REST

Para que o nosso sistema se comunique com o site (listagem de produtos), com o admin(crud), com aplicativos mobile(servidor), precisamos construir uma API (Application Public Interface), que é a forma com que os diversos clientes (site, admin, app mobile) possuem de interagir (pedir dados ou enviar informações) para o nosso banco de dados. Então, a API é o ponto de entrada para as demais formas de apresentar o conteúdo. É o backend em si da nossa aplicação.

Um sistema REST utiliza o protocolo http ou https (secure http, onde os dados trafegados são criptografados com ssl) para as requisições.
Nisso, algumas coisas estão bem definidas

Leitura complementar: https://blog.apigee.com/detail/restful_api_design

Exemplos de APIs

Twitter, Facebook, Google Maps, Sabesp API

Verbos http

Os verbos http nos possibilitam utilizar recursos diferentes com uma mesma URL. Por exemplo: /users/id se chamada com o verbo PUT pode atualizar esse usuário, mas /users/id se chamada com o verbo DELETE irá exclui-lo do banco de dados, e ainda se /users/id for chamada com o verbo GET a nossa aplicação irá retornar o usuário correspondente ao ID que informarmos.

Nossa aplicação irá utilizar os 4 seguintes verbos http:

- `POST   -> Create`
- `GET    -> Retrieve`
- `PUT    -> Update`
- `DELETE -> Delete`

Existem outros como PATCH, mas não iremos implementá-lo por enquanto.

Status Code

http://httpstatusdogs.com
https://www.flickr.com/photos/girliemac/sets/72157628409467125

- 200 - Ok
- 201 - Created
- 301 - Permanent redirect
- 404 - Not Found
- 500 - Internal server error

Fica definido por boa prática REST, que os endpoints (rotas) serão:

- escritas em minúsculo;
- separadas com hífen quando necessário;
- no plural;
- bem descritiva e concisa;
- utilizará o verbo http mais adequado;
- retornará um status code correspondente a resposta;

Podemos utilizar os seguintes comandos para fazer requisições http direto do nosso terminal:

POST (create)

```
$ curl -H "Content-Type: application/json" \
  -d '{"name":"Jane Doe"}' http://127.0.0.1:3000/products
```

GET (retrieve)

```
$ curl -H "Content-Type: application/json" \
  http://127.0.0.1:3000/products
$ curl -H "Content-Type: application/json" \
  http://127.0.0.1:3000/products/55060ceba8cf25db09f3b216
```

PUT (update)

```
$ curl -H "Content-Type: application/json" \
   -H "X-HTTP-Method-Override: PUT" -d '{"name":"Foo Bar"}' \
  http://127.0.0.1:3000/products/55061dc648ccdc491c6b2b61
```

DELETE (delete)

```
$ curl -X POST -H "Content-Type: application/json" \
  -H "X-HTTP-Method-Override: DELETE" \
  http://127.0.0.1:3000/products/55061dc648ccdc491c6b2b61
```

Ou então utilizarmos programas como o Postman.

Estrutura

Uma requisição http possui as seguintes partes:

- **query;**
- **parâmetro;**
- **cabeçalho;**
- **tipo (verbo);**
- **dados (corpo da requisição);**

Query é a parte da requisição que vai em formato de query string:
`?id=42&page=3`

Parâmetro é a parte da requisição enviada na URL:
`/products/`**`55061dc648ccdc491c6b2b61`**
Nesse caso, a string **55061dc..** é o parâmetro.

Cabeçalho são informações extras enviadas como por exemplo, qual o tipo de requisição estamos enviado
`-H "Content-Type: application/json" \`
Na linha acima informamos que a nossa requisição vai com um JSON,

Tipo é o verbo em si que iremos utilizar
`-H "X-HTTP-Method-Override: PUT"`
`-H "X-HTTP-Method-Override: DELETE"`

Note que estamos enviando em forma de cabeçalho o verbo. Para os servidor reescrever e entender que foi um PUT ou um DELETE. Isso é especialmente necessário, pois o html apenas implementa os verbos GET e POST, então para que o servidor entenda que estamos enviando algum outro diferente desses dois, o servidor precisa utilizar ou um cabeçalho ou um dado no meio do corpo da requisição para sobrescrever o tipo e entendê-la.

Dado é o corpo da requisição, ou seja, os dados que queremos enviar.
`-d '{"name":"Foo Bar"}'`
No nosso caso, será o produto em si, os inputs da nossa tag formulário que formarão o corpo da requisição.

ExpressJS

O ExpressJS (http://expressjs.com) será o framework web que utilizaremos para iniciar nosso servidor e gerenciar nossas rotas.

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

Agora, vamos criar o arquivo app.js que terá exatamente o código de exemplo da documentação do Express:

http://expressjs.com/starter/hello-world.html

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

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

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

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

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

No NodeJS quando precisamos utilizar um módulo baixado com o npm, utilizamos a função `require()`.
Na linha seguinte, `var app = express();` é onde iniciamos de fato o express.

```
app.get('/', function() {});
```

Já é a declaração de uma rota. No caso uma rota do verbo http GET, para a index, ou seja: http://localhost:3000/
Logo após, declaramos o servidor, ouvindo a porta 3000, e colocamos no terminal (console) uma mensagem informando o endereço e a porta.

Assim, já podemos iniciar, digitando no terminal

```
$ node app.js
```

E o servidor está funcionando. Quando visitarmos pelo browser o endereço: http://localhost:3000/, será mostrada a string Hello World!

Se quisermos mostrar um JSON na tela, podemos trocar o `res.send()` por
`res.json({ 'name': 'William Bruno', 'email': 'wbrunom@gmail.com' });`

E poderíamos também concatenar com um status code:

```
res.status(201).json({ 'name': 'William Bruno', 'email':
'wbrunom@gmail.com' });
```

Uma das vantagens de trabalhar com NodeJS é que como tudo é JavaScript, fazer uma API que retorne um JSON é perfeitamente natural, pois JSON é JavaScript (Javascript Object Notation), então não precisamos fazer um parser ou converter uma array para JSON, podemos escrever o JSON que queremos diretamente.

Express Generator

Caso você se sinta confortável já, por ter entendido o que é o NodeJS e como o ExpressJS funciona, vale a pena utilizar o módulo express-generator (instale ele globalmente com o npm), para que ele crie a estrutura inicial do projeto para você. Dentre outras coisas (pastas e arquivos) ele irá criar um código de tratamento de erros da forma correta.

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handlers

// development error handler
// will print stacktrace
/* istanbul ignore else */
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
     res.status(err.status || 500).json({ err: err.message });
  });
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
     message: err.message,
     error: {}
  });
});
```

Importante que o código de tratamento de erros seja declarado depois da criação de todas as rotas da aplicação. Pois se alguma requisição vier que não coincidir com as declaradas,

podemos seguramente assumir que o nosso servidor não tem aquele recurso e dispararmos um status code 404 para o cliente

```
  var err = new Error('Not Found');
  err.status = 404;
```

E caso seja algum outro tipo de erro, tratamos para apenas renderizar uma página html ou mostrar um JSON simplificado com o motivo de ter falhado, mas sem vazar para o cliente todo o stack trace do erro (detalhes que podem comprometer a segurança da aplicação como por exemplo, o caminho no servidor onde os nossos arquivos estão).
Caso ainda não queira utilizar o express-generator, copie ao menos as linhas acima do tratamento de erro, assim, a sua aplicação saberá o que fazer quando algum recurso não for encontrado, ou alguma coisa disparar um erro interno.

Nodemon

Para não ficarmos sempre parando o servidor com Ctrl + C e iniciando novamente com node app.js, cada vez que alterarmos um arquivo, iremos utilizar o módulo nodemon.

```
$ npm install -g nodemon
```

Ele fica “ouvindo” as alterações nos arquivos, e assim que um arquivo .js da nossa aplicação NodeJS for alterado, ele reiniciará o servidor para nós, agilizando bastante o nosso desenvolvimento. Agora, em vez de iniciarmos o servidor com node app.js, iremos usar

```
$ nodemon app.js
```

Swig

Caso quiséssemos renderizar um html em vez de mostrar uma string ou um json, teremos que configurar um sistema de template, e trocar o res.send() ou res.json() por res.render().

```
app.get('/', function (req, res) {
  res.render('index', { text: 'Lorem ipsum dolor sit amet.' });
});
```

O método .render do objeto res aceita dois argumentos. O primeiro é o nome do template a ser renderizado e o segundo é um objeto com variáveis que queremos injetar nesse html. Podemos usar para trocar o título da página, mostrar a mensagem numa página de erro…
E o nosso html para mostrar essa chave text é

```
  <p>{{ text }}</p>
```

Com dupla chaves. (Mustache style).

Para isso ser possível, precisamos avisar o ExpressJS qual engine de template iremos utilizar. E isso nós também configuramos dentro do app.js:

```
app.engine('html', swig.renderFile);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'html');
```

Servindo estáticos

Agora que realmente subimos um servidor NodeJS, não estamos nem abrindo o html direto com dois cliques e nem utilizando o apache, os nossos arquivos estáticos (*.css ou *.js), precisam também serem servidos através desse servidor, se não não serão encontrados. (veja o 404 no console do browser).
Utilizaremos o próprio ExpressJS para isso enquanto estamos desenvolvendo, mas na máquina de produção, o ideal é que deixemos esse trabalho para o nginx.

```
app.use(express.static(path.join(__dirname, 'public')));
```

E então o atributo href e src do css e js, ficarão a partir da raiz do servidor, já que dissemos que a raiz dos arquivos estáticos é a pasta public.
Vamos trocar `href="../public/css/all.min.css"` por `href="/css/all.min.css"`.
E o mesmo para o js.

```
<script type="text/javascript" src="/javascript/all.min.js"></script>
<link rel="stylesheet" type="text/css" href="/css/all.min.css" />
```

Lembrando que temos que reiniciar o servidor, parando ele com Ctrl + C e subindo novamente com node app.js, caso ainda não tenhamos colocado o nodemon que fará isso automaticamente para nós.

scripts do package.json

Dentro do arquivo `package.json` existe um objeto chamado `scripts`.
Nessa seção é onde configuramos a forma com que a nossa aplicação reage a comandos padrões (start, test), ou comandos personalizados que podemos criar e rodarão com npm.

É bem comum por questões de compatibilidade e de “esconder a complexidade”, encapsularmos a `nodemon app.js` dentro de `npm start`. Assim, independente de qual estratégia de start, ou quais outras coisas quisermos fazer no start, quem utilizar nossa aplicação, precisa conhecer apenas um simples comando.

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

Assim que rodarmos `npm start`, a linha `scripts.start` do `package.json` será chamada e o nodemon ouvirá nossa app.
Podemos criar outros comandos personalizados como:

```
  "scripts": {
     "start": "nodemon app.js",
     "test-api": "mocha tests/api/*",
     "test": "echo \"Error: no test specified\" && exit 1"
  },
```

E rodaríamos com:

```
$ npm run test-api
```

Ainda não iremos nos preocupar com os testes, para primeiro escrevermos as rotas e a conexão com banco de dados.

Method Override

Lembra que eu disse que o HTML só implementa GET e POST ?
Para prepararmos nossa app para entender os outros verbos, a partir de um cabeçalho ou um input enviado na requisição, iremos utilizar o módulo method-override.

```
$ npm install method-override --save
```

Fazemos o require dentro do app.js:

```
var methodOverride = require('method-override');
```

E depois declaramos as formas com as quais entendermos a sobrescrita de método:

```
app.use(methodOverride('X-HTTP-Method'));
app.use(methodOverride('X-HTTP-Method-Override'));
app.use(methodOverride('X-Method-Override'));
app.use(methodOverride('_method'));
```

Sendo a última _method aquela do input que comentei:

```
<input type="hidden" name="_method" value="PUT" />
```

Body Parser

Quando recebemos uma requisição no ExpressJS ela pode chegar em query string ou json. Por padrão o ExpressJS 4 não “entende” esses formatos, e recebe elas apenas como texto puro. Isso foi uma divisão de sub módulo que aconteceu na migração da versão 3 para a 4, onde retiraram diversas coisas do core e colocaram em módulos menores.

Então, para conseguirmos entender esses formatos precisamos do módulo body-parser.

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

Fazemos o require dentro do app.js:

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

E podemos informar para o express utilizá-lo:

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

Feito isso, o nosso servidor está pronto para trabalhar como uma API Rest.

O arquivo app.js

A esta altura você já deve ter começado a notar a importância do arquivo app.js.
É nele onde configuramos o nosso template engine, o method-override, o body-parser, onde ainda poderíamos incluir o controle de sessão, de cookie.. e tudo o que diz respeito a **configuração do servidor**.

Então a idéia é que fique centralizado tudo o que diz respeito a isso: configurar o server.

Para isso, algumas coisas como o controle de rotas e a inicialização em si do servidor, ficam melhor se movidas para outros arquivos. Deixando assim o app.js com uma única responsabilidade: **configurar**.

No momento o app.js está assim:

```
var express = require('express');
var path = require('path');
var swig = require('swig');
var methodOverride = require('method-override');
var bodyParser = require('body-parser');
var app = express();

// view engine setup
app.engine('html', swig.renderFile);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'html');

app.use(express.static(path.join(__dirname, 'public')));

app.use(methodOverride('X-HTTP-Method'));
app.use(methodOverride('X-HTTP-Method-Override'));
app.use(methodOverride('X-Method-Override'));
app.use(methodOverride('_method'));

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

app.get('/', function (req, res) {
  res.render('index', { text: 'Lorem ipsum dolor sit amet.' });
});

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

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

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

E vamos deixá-lo com essa cara:

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

```
var path = require('path');
var swig = require('swig');
var methodOverride = require('method-override');
var bodyParser = require('body-parser');
var app = express();

// view engine setup
app.engine('html', swig.renderFile);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'html');

app.use(express.static(path.join(__dirname, 'public')));

app.use(methodOverride('X-HTTP-Method'));
app.use(methodOverride('X-HTTP-Method-Override'));
app.use(methodOverride('X-Method-Override'));
app.use(methodOverride('_method'));

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

app.use('/', require('./routes')(app));

module.exports = app;
```

Note que mudamos as 2 últimas declarações. O `app.get()` virou `app.use()` e o `app.listen` virou um `module.exports`. Destaquei em negrito as linhas alteradas.

Criamos então o arquivo `www` dentro da pasta `bin`: `bin/www`

```
#!/usr/bin/env node

var app = require('../app');
var server = app.listen(3000, function () {

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

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

E alteramos lá no package.json a inicialização do servidor:

```
  "scripts": {
     "start": "nodemon bin/www",
     "test": "echo \"Error: no test specified\" && exit 1"
  },
```

E para as rotas, criamos o diretório `routes`, com o arquivo `index.js` com o seguinte conteúdo:

http://expressjs.com/guide/routing.html

```
module.exports = function(app) {
  var express = require('express'),
     router  = express.Router();

  router.get('/', function (req, res) {
     res.render('index', { text: 'Lorem ipsum dolor sit amet.' });
  });

  return router;
};
```

Nosso código até agora

https://github.com/wbruno/curso-nodejs-16h-novatec/tree/v0.2.0

Rotas

Agora que já extraímos o arquivo de rotas de dentro do `app.js`, podemos implementar os demais verbos http.

```
module.exports = function(app) {
  var express = require('express'),
     router  = express.Router();

  router.route('/products')
    .get(function(req, res, next) {
      res.send('GET ALL');
    })
    .post(function(req, res, next) {
      res.send('POST');
    });
```

```
  router.route('/products/:_id')
    .get(function(req, res, next) {
       res.send('GET ID');
    })
   .put(function(req, res, next) {
      res.send('PUT ID');
   })
  .delete(function(req, res, next) {
    res.send('DELETE ID');
  });

  router.get('/', function (req, res) {
    res.render('index', { text: 'Lorem ipsum dolor sit amet.' });
  });

  return router;
};
```

Usaremos a arquitetura MVC para desenvolver nossa API. Nos nossos arquivos de rotas ficarão apenas as declarações de endpoints (caminhos), e cada um invocará o seu controller correspondente.

Dentro de `routes/index.js` deixaremos apenas:

```
app.use('/products', require('./products'));
```

E então em `routes/products.js`:

```
var express = require('express'),
     router  = express.Router();

var ProductController = require('../controllers/ProductController');

router.route('/')
  .get(ProductController.getAll.bind(ProductController))
  .post(ProductController.create.bind(ProductController));

router.route('/:_id')
  .get(ProductController.getById.bind(ProductController))
  .put(ProductController.update.bind(ProductController))
  .delete(ProductController.delete.bind(ProductController));

module.exports = router;
```

Controllers

E com o controller conseguimos retirar das rotas a responsabilidade de lidar com o request e de devolver o response para o usuário.

Sendo o nosso controller:

```
function ProductController() {}

ProductController.prototype.getAll = function(req, res, next) {
  res.send('GET ALL');
};
ProductController.prototype.create = function(req, res, next) {
  res.send('POST');
};
ProductController.prototype.getById = function(req, res, next) {
  res.send('GET ID');
};
ProductController.prototype.update = function(req, res, next) {
  res.send('PUT ID');
};
ProductController.prototype.delete = function(req, res, next) {
  res.send('DELETE ID');
};

module.exports = new ProductController();
```

O verdadeiro responsável por entender o request, chamar o model e então responder o usuário com um response ou então redirecioná-lo com o `next()` para subirmos o tratamento de erro até o `app.js` onde ele será capturado.

Models

O model é a camada responsável pelo acesso aos dados. Apenas os models em toda a aplicação é que irão conhecer o banco de dados.

```
function ProductModel() {}

ProductModel.prototype.find = function(query, callback) {
};
```

```
ProductModel.prototype.create = function(data, callback) {
};
ProductModel.prototype.update = function(data, id, callback) {
};
ProductModel.prototype.delete = function(id, callback) {
};

module.exports = new ProductModel();
```

Agora, o nosso Controller pode usar o model:

```
function ProductController(Model) {
  this.Model = Model;
}
ProductController.prototype.getAll = function(req, res, next) {
  this.Model.find({}, function(err, result){
    if (err) {
      return next(err);
    }
    res.json(result);
  });
};
//..
module.exports = new ProductController(ProductModel);
```

Banco de dados

Utilizaremos o MongoDB como banco de dados da aplicação. Para isso, iremos construir um "wrapper" para conectar no mongo e fazer o handler de erros de conexão em um arquivo chamado `Mongo.js` dentro do diretório `db/`.

```
var mongojs = require('mongojs'),
     debug   = require('debug')('curso:db'),
     config  = require('config');

'use strict';
var db;

function _connection(env) {
  var username = config.get('mongo.username'),
     password = config.get('mongo.password'),
     server   = config.get('mongo.server'),
     port  = config.get('mongo.port'),
```

```
      database = config.get('mongo.database'),

      auth = username ? username + ':' + password + '@' : '';

  return 'mongodb://' + auth + server + ':' + port + '/' + database;
}

function _init(url) {
  debug(url);
  db = mongojs(url);
  db.on('error', function(err) {
     debug(err);
  });
}

_init(_connection(process.env));

module.exports = db;
```

Por boa prática, o módulo node-config (https://github.com/lorenwest/node-config) será utilizado para ter os dados de acesso do banco, então criarmos também o arquivo `config.json` dentro do diretório `config`.

```
{
  "mongo": {
     "server": "localhost",
     "username": "",
     "password": "",
     "port": 27017,
     "database": "curso"
  }
}
```

Com isso já podemos implementar de fato cada um dos métodos do **Model** e os seus correspondentes no **Controller**.

Nosso código até agora

https://github.com/wbruno/curso-nodejs-16h-novatec/tree/v0.4.0

Testes Funcionais

A idéia por trás dos **testes funcionais** é garantir que a nossa aplicação está funcionando e se comportando da forma com que esperamos. Para isso testamos conexão, escrita e leitura no banco de dados. Os **testes unitários** seriam aqueles em que faríamos um **mock** do banco, e então testaríamos o nosso código em si, função por função, e não a "funcionalidade" total.

Iremos escrever alguns testes funcionais para entendermos como funciona, e já termos algo automático que garante o funcionamento do sistema, para assim termos mais segurança caso queiramos refatorar alguma coisa.

Nosso framework de testes será o **mocha** (https://github.com/mochajs/mocha) e como estamos escrevendo um teste "caixa preta" (batendo nas rotas da API, sem necessariamente conhecer o código fonte, mas sim o comportamento esperado), utilizaremos também o módo **supertest** (https://github.com/visionmedia/supertest) para fazer as requisições. Ambos serão instalados como dependências de desenvolvimento:

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

Quanto as verificações, optei pelo módulo nativo do NodeJS, o **assert** (https://nodejs.org/api/assert.html), até para não incharmos nossa aplicação de dependências e módulos de terceiros, utilizaremos ao máximo os recursos da própria plataforma.

Mocha

```
//tests/api/products_test.js

describe('Products Endpoints', function() {

});
```

A função `describe` do mocha possui como objetivo "organizar" nossos testes. Ela recebe como argumentos uma string que é a descrição em si da suite de testes que iremos escrever e uma função, dentro da qual iremos declarar nossos testes.

Vamos escrever um arquivo de testes por rota e colocaremos um `describe` para cada um deles, assim a execução será algo como:

Products Endpoints

- ✓ GET /products | returns all products
- ✓ GET /products/:_id | returns a product
- ✓ POST /products | create a new product
- ✓ PUT /products/:_id | update a product
- ✓ DELETE /products/:_id | remove a product

Cada teste é declarado com uma função `it()`, que possui uma assinatura bem semelhante ao `describe`, pois também recebe como primeiro argumento uma string e como segundo uma função.

```
var request = require('supertest'),
      app = require('../../app');

describe('Products Endpoints', function() {
  it('GET /products | returns all products', function() {
  });

  it('GET /products/:_id | returns a product', function() {
  });

  it('POST /products | create a new product', function() {
  });

  it('PUT /products/:_id | update a product', function() {
  });

  it('DELETE /products/:_id | remove a product', function() {
  });
});
```

Nossos testes são executados com o comando:

```
$ node_modules/mocha/bin/mocha tests/api/*
```

Para facilitar, vamos colocá-lo dentro do `scripts` no `package.json`.

```
"scripts": {
      "start": "export DEBUG=curso:* && nodemon bin/www",
      "test": "export DEBUG=curso:* && node_modules/mocha/bin/mocha
tests/api/*"
},
```

Feito isso, podemos executar com apenas:

```
$ npm test
```

E utilizando o supertest, com o módulo debug para entendermos o que volta na requisição:

```
it('GET /products | returns all products', function(done) {
  request(app)
    .get('/products')
    .expect(200)
    .end(function(err, result) {
     var data = result.body;
     debug(err, data);

     done();
    });
});
```

Assim será para todos os demais `it()`. O mocha está pronto para testar funções assincronas. Basta invocar a função done() que recebemos como argumento do callback do it:

```
it('string', function(done) {
   //something assync
  done();
});
```

beforeEach()

Como o comportamento ideal é que um teste não interfira no outro, então implementaremos um hook beforeEach() que se encarregará de apagar os registros entre um teste e o próximo.

```
beforeEach(function(done){
  mongo.collection('products').remove({}, done);
});
```

E nos testes que esperam que haja algo no banco, inserimos diretamente:

```
it('GET /products/:_id | returns a product', function(done) {
  mongo.collection('products').insert({ name: 'Jane Doe' },
function(err, result) {
     var _id = result._id.toString();
```

```
    request(app)
      .get('/products/' + _id)
      .expect(200)
      .end(function(err, result) {
        var data = result.body;

        assert.deepEqual(data, { _id: _id, name: 'Jane Doe' });
        done();
    });
  });
});
```

Nosso código até agora
https://github.com/wbruno/curso-nodejs-16h-novatec/tree/v0.5.0

Node Debug

Durante o desenvolvimento com NodeJS podemos ter dúvidas sobre o que veio na requisição, sobre o que voltou do banco de dados, ou apenas querer dar uma olhada mais próxima de um objeto ou propriedade.

Para isso, podemos assim como no JavaScript client side, utilizar a função `console.log()`. Inclusive já devemos ter utilizado ela algumas vezes durante o curso. Porém, é uma má prática deixar `console.log()'s` perdidos pela aplicação quando formos colocá-las em produção. Por um simples motivo: tudo o que escrevermos com `console.log()` no terminal, irá para o arquivo de log da aplicação, caso configuremos para o node subir gravando log. E por isso, não deixaremos “debugs” aleatórios num arquivo tão importante como esse.

Por isso, utilizaremos o módulo debug https://github.com/visionmedia/debug
Ele trabalha dependendo de uma variavel de ambiente chamada **DEBUG**. Se ela estiver setada, o módulo irá imprimir na tela os debugs, caso não, não irá. Assim não precisamos ficar preocupados em sair retirando da aplicação `console.log()` pois usaremos apenas `debug()`.

```
$ npm install --save debug
```

E para utilizar, ele trabalha com um sistema de namespaces para organização. Tendo assim o nome da nossa aplicação e o “módulo” dela que queremos debugar:
Faça o require dentro do arquivo .js que quiser debugar:

```
var debug = require('debug')('curso');
```

E para utilizar:

```
debug('Hi!');
```

Lembrando da variavel de ambiente DEBUG. Podemos pedir para que o debug mostre tudo:

```
$ export DEBUG=*
```

E nesse caso, veremos até o debug do ExpressJS, algo mais ou menos assim:

express:router use / query **+1ms**
express:router:layer new / **+0ms**

E para vermos apenas o debug da nossa aplicação:

```
$ export DEBUG=curso:*
```

Pois esse foi o namespace que declaramos no momento do require:
`require('debug')('curso');`

Que poderia ser também:
`require('debug')('curso:model')`, `require('debug')('curso:router')` ou
`require('debug')('curso:controller')`

Depende de como queremos organizar. Uma outra feature bem legal, é que ele mostra o tempo decorrido entre duas chamas do método debug. Então para medir a performance isso é bem útil. Igualzinho faríamos com o `console.time()` e `console.timeEnd()`.

Forever

Precisamos ter certeza que a nossa aplicação continue rodando quando formos colocá-la em produção. Para isso é importante ter um processo monitorando a atividade dela, que a reinicie automaticamente caso alguma exceção por algum motivo a derrube (crash).

O módulo que eu gosto de usar é o forever https://github.com/foreverjs/forever
Instalaremos ele globalmente:

```
$ npm install forever -g
```

E então usaremos o forever para subir em produção, enquanto usamos o nodemon em desenvolvimento:

```
$ forever start /var/www/curso/bin/www
```

Feito isso, alguns dos métodos que temos disponíveis são: stop, list e restart

```
$ forever stop /var/www/curso/bin/www
$ forever list
$ fover restart /var/www/curso/bin/www
```

As outras versões são: `stopall` e `restartall`.

NGINX

Não deixamos o NodeJS servidor diretamente na porta 80 por motivos de segurança, como não deixar o NodeJS rodar diretamente como usuário com permissões root.

Então o Nginx que é um servidor web ficará na frente do node. Fazendo um proxy da porta 3000 que o node serve para a porta 80 que é aquela que fica aberta para aplicações web http.

```
upstream nodejs  {
  server 127.0.0.1:3000;
}

server {
  listen 80;
  server_name site.com.br www.site.com.br;
  root /var/www/site.com.br/;

  if ($http_host != "site.com.br") {
     rewrite ^ http://site.com.br$request_uri permanent;
  }

  error_page 404 500 502 503 504 /50x.html;

  location /50x.html {
     internal;
  }

  location / {
```

```
      proxy_set_header X-Real-IP  $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $host;
      proxy_pass http://nodejs;
  }

  location ~*
\.(?:ico|css|js|gif|jpe?g|png|ttf|woff|svg|eot|xml|txt)$ {
      access_log off;
      expires 30d;
      add_header Pragma public;
      add_header Cache-Control "public, mustrevalidate,
proxy-revalidate";
      root /var/www/site.com.br/public;
  }

  location ~ /\.ht {
      deny all;
  }
}
```

Unix Service

Mais uma camada de abstração que iremos adicionar será um unix service. Assim, caso desejemos mudar do forever para o **pm2** (https://github.com/Unitech/pm2) ou qualquer outra estratégia de monitoramento do processo NodeJS, não precisamos alterar nada no nosso CI (Travis, Jenkins) se ele que fizer nosso deploy.

O nome que você escolher para ele será o nome do serviço:

```
$ service nodejs
$ service nodejs start
$ service nodejs stop
$ service nodejs status
```

Este arquivo ficará dentro da pasta `/etc/init.d`

```
#!/bin/sh

### BEGIN INIT INFO
# Provides:          site
# Required-Start:    forever node module
```

```
# X-Interactive:     true
# Short-Description: Site initscript
# Description:       Uses forever module to running the application
### END INIT INFO

NODE_ENV="production"
PORT="3000"
APP_DIR="/var/www/site.com.br"
NODE_APP="bin/www"
CONFIG_DIR="$APP_DIR/config"
LOG_DIR="/var/log/site"
LOG_FILE="$LOG_DIR/app.log"
NODE_EXEC="forever"

###############

USAGE="Usage: $0 {start|stop|restart|status}"

start_app() {
  mkdir -p "$LOG_DIR"

  echo "Starting node app ..."
  PORT="$PORT" NODE_ENV="$NODE_ENV" NODE_CONFIG_DIR="$CONFIG_DIR"
forever start "$APP_DIR/$NODE_APP" 1>"$LOG_FILE" 2>&1 &
}

stop_app() {
  forever stop "$APP_DIR/$NODE_APP"
}

status_app() {
  forever list
}

restart_app() {
  forever restart "$APP_DIR/$NODE_APP"
}

case "$1" in
  start)
     start_app
  ;;
```

```
  stop)
      stop_app
  ;;

  restart)
      restart_app
  ;;

  status)
      status_app
  ;;

  *)
      echo $USAGE
      exit 1
  ;;
esac
```

Heroku

Caso utilizemos um servidor no modelo PaaS (Plataform as a Service), não vamos precisar instalar o NodeJS no Linux, nem configurar o Nginx e nem do unix service. O host nos fornecerá tudo isso. Só iremos configurar tudo isso na mão, se optarmos por um Iaas (Infrastructure as a Service), pois nesse modelo recebemos uma máquina limpa, sem nada instalado, apenas a distribuição Linux que escolhemos.

O Heroku possui um PaaS para NodeJS muito bom: http://heroku.com/
E podemos utilizá-lo de graça para subir nossas aplicações de teste.

Deploy

Dentre as diversas alternativas para deploy, vamos fazer nesse instante com um git pull, e depois você escolhe qual melhor se adequa ao seu perfil e ao host que você utiliza.
http://imasters.com.br/front-end/javascript/configurando-nodejs-em-producao-em-um-cloud-server/