04

Instalação do Node e configuração do Express

Olá! Bem vindos ao curso de Node.js e api REST da Alura!

Eu sou o Júlio e serei seu instrutor nessa jornada de aprendizado de bastante javascript e muitas ferramentas e pacotes do universo do Node.

Vamos então começar explicando um pouco o projeto a ser desenvolvido e o porque da escolha do Node como ambiente de desenvolvimento.

Começando o projeto

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

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

Nesse curso, você vai aprofundar seus conhecimentos na plataforma Node e entender quais as principais práticas e bibliotecas utilizadas pelo mercado e desenvolvidas pela comunidade para resolver problemas que comumente surgem no desenvolvimento de sistemas web e que tem intenção de funcionar como servidores e consumidores de outros serviços.

Criação do projeto

Neste curso desenvolveremos uma aplicação para funcionar como um *gateway* de pagamentos, ou seja, uma plataforma que pode ser utilizada por outros aplicativos para integrar meios de pagamento. Esse domínio foi escolhido por proporcionar diversas necessidades de implementação relevantes para o conteúdo do curso e por ter uma inspiração real que funcionamuito bem e é amplamente reconhecida no mercado, que é a *API* para integrações do **PayPal**.

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

Rodando o primeiro código

A primeira tarefa é instalar o Node.js. Esse é um processo até certo ponto simples, podemos simplesmente seguir os passos descritos na home da página no node https://nodejs.org/ (https://nodejs.org/).

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

Note que nessa página já são mostradas as versões mais recentes, tanto a *LTS (Long Therm Support)*, quanto a que possui as *features* mais recentes, que tende a estar sempre algumas versões a frente por razões óbvias. Para este curso utilizaremos a

LTS, que é a mais provável de se usar em um ambiente real de produção.

Caso você prefira ver outras opções de download, inclusive para diferentes plataformas, pode visitar também a página de downloads do Node https://nodejs.org/en/download/ (https://nodejs.org/en/download/).

Uma vez com o Node devidamente instalado, podemos verificar se está tudo ok, executando um comando que mostra a versão dele, por exemplo.

Para isso, basta abrir uma janela qualquer do terminal e digitar o comando a seguir:

```
node --version
```

Criação do projeto com o Express.js

Para criação de um projeto em Node.js, o procedimento é bastante simples e já conhecido por quem fez o primeiro curso. O primeiro passo é criar uma pasta para o projeto *PayFast*, que deverá ter esse mesmo nome e em seguida será criado o projeto a partir do comando `npm init`. O passo a passo fica assim:

```
mkdir payfast
cd payfast
npm init
```

A sigla *npm* é um acrônimo para *Node Package Manager* e ele é o gerenciador de pacotes oficial do Node.js. O npm já vem instalado quando o Node é instalado na máquina e ele tem a função de instalar, compartilhar e distribuir o código; gerenciar dependências no projeto; e até compartilhar e receber feedbacks de outras pessoas. Veremos mais sobre utilizaremos bastante essa ferramenta ao longo deste curso.

Após a execução do comando `npm init`, o Node exibe uma sequência de perguntas no terminal para usar como informações do projeto, como o nome que queremos dar a ele, qual a versão, uma descrição, o nome do autor e outras. A princípio deixaremos todas com os valores default e ao final da execução será exibido no terminal um *json* que guarda esses dados. Ele é o **package.json** e ficará com um formato parecido com o seguinte:

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

Para suportar todas as características básicas de uma aplicação web sem que tenhamos que nos preocupar em ficar implementando coisas repetitivas, usaremos o **Express**. Ele é framework web compatível com as API's fornecidas pelo Node.js. Dentre os benefícios trazidos por ele podemos citar:

- isolar o código de tratamento de diferentes urls.

- isolar a escrita do código html da lógica da aplicação.
- responder conteúdos com formatos diferentes, baseado nas informações do request.
- lidar com os métodos de envio de dados, por exemplo requisições feitas com GET e com POST.

Criando um novo projeto com o Express

Para poder utilizar o *Express* ou qualquer outra lib externa em projeto Node, é preciso que seja realizada a instalação desse novo pacote. Essa instalação depende de baixar o pacote e suas dependências. Portanto, para facilitar o trabalho utilizaremos o npm, que já sabe exatamente onde buscar e armazenar todo esse código.

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

Esse formato de instalação com o npm também já é conhecido e será repetido várias vezes ao longo do curso, mas ainda assim vale reforçar o que cada comando está fazendo: o `install` obviamente indica que algo deve ser instalado; a seguir passamos o nome do pacote a ser instalado, no nosso caso *express*, para que o npm busque em seus repositórios de pacote; por fim o parâmentro `--save` serve para que o próprio npm altere o nosso arquivo *package.json* e já deixe explícito que o projeto possui essa dependência.

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

Como em todo projeto de software, precisamos organizar os arquivos de uma maneira que favoreça a evolução e manutenção do código. Por já ter alguma experiência com o projeto do primeiro curso, vamos seguir uma linha semelhante.

Sabemos que um projeto Node precisa de um arquivo que inicie a aplicação. É bastante comum que esse arquivo seja criado na raíz do projeto e receba um nome que descreva bem sua função. Seguiremos estes preceitos e nomearemos nosso arquivo como **index.js**. Para criá-lo basta então executar algum comando de criação de arquivos no terminal, dentro da pasta do projeto ou criá-lo já dentro de alguma IDE. Aqui utilizaremos o terminal:

```
touch index.js
```

O `index.js` precisa iniciar o servidor e para isso vamos utilizar o Express, que definimos como o framework web da aplicação. Poderíamos fazer essa implementação da seguinte forma:

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

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

Na primeira linha carregamos o módulo do Express, que foi previamente instalado com o npm, para dentro de uma variável chamada `express`. Em seguida declaramos uma nova variável, chamada `app` e fizemos ela receber a invocação do módulo

que havia sido carregado na primeira variável. Dessa forma fizemos com que a variável `app` passasse a armazenar o **objeto** do Express de fato.

A partir desse objeto pudemos invocar a função que `listen`, que inicia o servidor conforme queríamos em uma porta determinada via parâmetro, no nosso caso definimos a porta 3000, mas poderia ser qualquer outra que não estivesse ocupada.

Essa função também recebe como parâmetro uma **função de callback** para que possamos indicar o que deve acontecer uma vez que a execução da função `listen` tenha sido completada com sucesso. Uma simples mensagem no console informando que o servidor está rodando foi suficiente.

Como é sabido, as funções de callback são muito utilizadas no mundo do Node e nesse curso também faremos bastante uso delas, mas também mostraremos uma outra forma de lidar com a capacidade assíncrona do Node, sem ter que necessariamente fazer uso dos callbacks.

Agora basta executar o arquivos `index.js` para que o servidor passe a rodar de fato.

```
node index.js
```

Após a execução do comando acima, no terminal, deve ser exibido no próprio terminal a frase **"Servidor rodando!"** que foi colocada no `console.log` no *callback* da função `listen`. Basta agora acessar no navegador a url *http://localhost:3000 (http://localhost:3000)* e veremos que o servidor está de fato rodando e atendendo requisições na porta 3000. Porém é exibida a mensagem *"Cannot GET /"*. Isso acontece porque obviamente ainda não foi definida nenhuma rota para o projeto. Esse ponto será resolvido logo a seguir.

Organizando o projeto e separando as responsabilidades

Um primeiro ponto bastante importante é entender qual é a função de cada arquivo, ou cada, unidade de código, dentro do projeto. O arquivo `index.js`, por exemplo, atualmente está executando duas funções: iniciar o objeto do express e iniciar a própria aplicação do *PayFast*. Mas sabe-se que o ideal é que cada arquivo tenha uma única responsabilidade e saiba fazê-la muito bem e de forma simples.

Para isso, vamos então isolar o carregamento do objeto do **Express** em um arquivo específico. Essa decisão ajudará bastante também na evolução do código, pois o objeto do **Express** terá algumas outras configurações mais específicas e portanto, ficaria realmente difícil dar manutenção nesse código caso ele estivesse misturado com o início da aplicação.

Para que os arquivos do código também fiquem organizados de maneira clara para os programadores, faz sentido que eles fiquem agrupados em pastas de acordo com suas funcionalidades em comum. Nesse caso, criaremos então uma pasta chamada `config`, que ficará responsável por todos os arquivos de código relacionados à configuração da aplicação, ou seja, código que não necessariamente seja de regra de negócio. E dentro dessa pasta será criado o arquivo `custom-express.js`, que ficará com o seguinte código:

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

module.exports = function() {

    var app = express();

    return app;
}
```

Basicamente um código que isola a criação do **Express**. Porém, este arquivo é bastante importante e terá várias outras funções. A primeira delas será a primeira configuração extra que será feita no **Express** e sua justificativa vem da necessidade de centralizar todos os carregamentos de módulos que o projeto precisará utilizar num só lugar. Para conseguir esse objetivo foi criado o projeto **consign**, que é considerado o sucessor do projeto que foi utilizado no primeiro curso, o **express-load**.

Como já é de praxe, essa nova *lib* será adicionada ao projeto através do *npm*:

```
npm install consign --save
```

O melhor local para centralizar o carregamento de módulos relativos ao express é justamente no arquivo onde ficou isolada sua configuração. Isso implica na seguinte mudança no `custom-express.js` :

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

module.exports = function() {

    var app = express();
    consign().into(app);

    return app;
}
```

O módulo foi carregado e em seguida foi invocada a função retornada por ele. E por último foi determinado através da função `into()` que tudo o que for carregado pelo **consign** deve ser incluido ao objeto do **express** que está sendo referenciado pela variável **app**.

O arquivo `index.js` agora precisa somente carregar o arquivo `custom-express.js` através da função *require* e este arquivo já retorna o objeto do **express** para que fique guardado em uma variável. Neste caso a variável que guarda o **express** manteve o seu nome anterior que era **app**.

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

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

Rotas e Controllers com Express

Agora que a inicialização do projeto e o carregamento dos seus módulos estão bem definidos, pode-se partir para a parte mais interessante do código, onde serão definidas de fato as funcionalidades que serão implementadas na aplicação *PayFast*.

Como a aplicação representa um serviço web, cada funcionalidade será definida através de uma **rota**, que nada mais é que definir para cada serviço a ser implementado, uma *url*, a partir da qual esse serviço será acessado, uma função de *callback* que deverá conter o código a ser executado quando esse serviço for requisitado, e o verbo *HTTP* que define a forma de acesso a essa rota.

Para continuar evoluindo o código de maneira organizada, pode-se criar uma novas pasta para guardar somente os arquivos que definem rotas. Como esses arquivos são também conhecidos no mundo *web* como os *controladores* da aplicação, esta nova pasta será chamada **controllers** e dentro dela deverá criado o novo arquivo `pagamentos.js`

```
module.exports = function(app) {
   app.get("/pagamentos",function(req, res) {
        res.send('ok');
    });
}
```

Este código define que a partir de agora o *PayFast* sabe atender uma requisição que chegue em seu contexto para a *url* **/pagamentos** e a partir de uma chamada com método **GET** do **HTTP**. A funcionalidade implementada por essa primeira rota é simplesmente enviar a resposta "ok" para o cliente que a requisitou.

Note que esse códiog utiliza uma variável **app** sendo recebida como parâmetro da função que foi atribuida ao `module.exports` , porém essa variável não foi declarada em local nenhum do arquivo, mas o código funciona! Isso é possível porque o **express** é que está gerenciando a execução desse código e ele sabe que essa função espera como parâmetro a referência para o seu próprio objeto.

Até aí tudo bem, mas o arquivo que inicia o **express** é o `index.js` e ele não conhece este novo arquivo `controllers/pagamentos.js` que acabou de ser criado. Como é possível então que essa rota que acabou de ser definida nesse ponto seja conhecida pelo arquivo que iniciou o servidor e está escutando as requisições?

Aqui é que o **consign** passa a ajudar bastante! Caso ele não fosse utilizado, o que teria que ser feito seria fazer um **require()** do arquivo `pagamentos.js` dentro do arquivo `index.js` e assim deveria ser feito para todos os novos *controllers* que fossem criados, assim como quaiquer outros arquivos que precisem ser conhecidos pelo **express**.

Com o **consign**, basta ensinar ao **express** já no seu carregamento que ele deve conhecer todos os arquivos de uma determinada pasta. Isso é feito informando a pasta como parâmetro no momento do carregamento do **consign**.

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

module.exports = function() {

    var app = express();
    consign()
         .include('controllers')
        .into(app);

    return app;
}
```

Agora o **consign** sabe informar ao **express** que ele deve fazer um *scan* de todos os arquivos dentro da pasta `controllers` carregá-los para dentro do seu objeto.

O mundo **Node.js** já possui várias soluções bastante conhecidas e utilizadas na comunidade para resolver problemas que são comuns a muitos desenvolvedores, como essa que acaba de ser apresentada, e várias outras dessas serão utilizadas ao longo do curso para entender importantes conceitos e implementar funcionalidades muito úteis para o mercado.