07

Assim nasce nosso servidor!

Instalou a infraestrutura (Node.js e MongoDB) necessária e baixou o projeto como descrito nos exercícios obrigatórios do capítulo anterior? Se sim, vamos começar os trabalhos.

"Parindo" nosso servidor

Tenho a pasta do projeto `alurapic` descompactada na minha área de trabalho. Dentro dela, tenho a pasta `public` com nosso projeto Angular que sequer podemos rodar. Qual motivo? Bem, não há qualquer código no projeto que suba um servidor, muito menos que acesse um banco de dados. Precisamos criar nosso servidor!

No mundo Node.js, também criamos arquivos com a terminação `.js`. E se queremos criar um servidor, nada mais justo do que criarmos um arquivo de mesmo nome com o prefixo tão conhecido por aqueles que programam Javascript.

Como de costume, utilizarei o Sublime Text Editor, que é o meu editor favorito. Não sei qual é o seu editor, mas este me permite abrir a pasta do projeto e visualizar todo seu conteúdo sem eu precisar sair dele. Excelente não?

Agora que estou com a pasta aberta, criarei na raiz do projeto, isto é, dentro da pasta `alurapic`, meu arquivo **server.js**. A grande questão agora é como vamos parir um servidor neste arquivo? Se você veio do mundo Java ou C#, na plataforma Node.js não temos um servidor separado no qual rodamos nossa aplicação. Nossa aplicação é também o próprio servidor! Diferente? Mas criar um servidor na mão deve ser muito complexo, não? "Sigam-me os bons!".

Vamos criar um servidor ftp, http ou de email? Queremos um servidor http, claro. Nada mais justo do que criarmos uma variável de mesmo nome:

```
// alurapic/server.js

var http;
```

Legal, na plataforma Node.js criamos variáveis da mesma forma que criamos em nosso navegador, mas ainda não atribuímos qualquer valor para a variável. Nossa variável **requer** alguém que saiba criar um servidor http. Onde está esse alguém, eu te pergunto. Se ele não existe, precisaremos criar um, certo? Errado, imagine o tempão que vou gastar programando esse "alguém do além". Mas nem tudo está perdido.

No mundo Node.js, esse "alguém" que requeremos é um módulo. Um módulo, Flávio? Sim, porque um módulo é uma unidade isolada de código com um ou mais arquivos `.js`, que expõe para seu utilizador apenas o que é necessário. Fazendo uma analogia, pense um módulo como uma peça de lego. Cada peça tem uma função bem definida, inclusive podemos combinar várias peças para criar algo maior, mas cada peça continua sendo aquela peça, sem interferir na da outra.

A instalação do Node.js já traz por padrão um módulo especializado na criação de servidores e seu nome é **http**. Muito original, não? Mas você pode estar se perguntando: "Flávio, entendi que o Node.js vem com esse módulo e talvez muitos outros, mas como faço para ter essa funcionalidade de **requerer** módulos no meu código?". Basta traduzir a palavra "requerer" para o inglês que você encontrará essa funcionalidade. Como fica?

```
// alurapic/sever.js

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

A função **require** é aquela do Node.js responsável em carregar um módulo dentro de outro. Como assim dentro de outro? Todo arquivo `.js` que criamos no Node.js é um módulo. Em nosso exemplo, o módulo `server.js` foi o que criamos e ele está requerendo o módulo `http` do Node.js que já existe.

Todo módulo sabe fazer algo e podemos pedir carinhosamente para ele: http, por favor, crie um server para mim? *Do you speak english?* Se sabe falar um pouco de inglês, basta traduzir *crie um servidor* para o inglês que você já sabe qual o nome da função de `http` responsável pela criação do servidor:

```
// alurapic/sever.js

var http = require('http');

http.createServer();
```

Sem orelha, como ele pode nos ouvir?

Flávio, você está de brincadeira? Só essa linha de código cria um servidor na plataforma Node? De brincadeira não estou, mas isso ainda não é suficiente. Todo servidor precisa **escutar** uma porta. Há servidores web que escutam na porta 80, 8080, inclusive a famigerada porta 171 (você confiaria?). Precisamos indicar para nosso recém parido server qual porta queremos que ele escute. Em nosso caso, no ambiente de desenvolvimento é muito comum a porta `3000` :

```
// alurapic/sever.js

var http = require('http');

http.createServer().listen(3000);
```

Gostou? Encadeamos a função `listen` ("escutar" em português) que fará nosso servidor monitorar a porta 3000. O que significa isso? Qualquer requisição feita para `localhost:3000` fará com que nosso servidor entre em ação. Quer dizer que um servidor pode ter mais de uma porta? Claro, é por isso que em um mesmo servidor temos uma porta para FTP, SSH, HTTP etc. Mas se você chegou neste treinamento, eu imagino que você já deva conhecer esses detalhes de funcionamento de um server, certo? Muito bem, nosso servidor nada faz, mas será que ele já consegue escutar quando alguém o está acessando? Vamos fazer um teste?

Nosso bebê não escuta... É surdo?

Quando você trabalha com a plataforma Node.js, é necessário ter traquejo no terminal/cmd/prompt de comando da sua plataforma. Eu tenho manejo com o terminal do MAC, você tem com seu terminal do Windows? Eu vou abrir o meu terminal e mudar para o diretório do meu projeto. Sempre que você abrir seu terminal até o final do curso será para estar dentro da pasta do projeto, por isso sempre tenha certeza que esteja lá.

Agora que estou dentro de `alurapic` , para subirmos nosso servidor, basta chamarmos o interpretador do node seguido do nome do nosso arquivo, porém não há a necessidade de passarmos a extensão `.js` , ela é opcional:

```
node server
```

Se você olhar bem, nosso terminal fica travado e não recebemos nenhuma mensagem. Bom, travar o terminal é normal, pois o servidor ficará rodando a vida toda até que ele seja terminado fechando o terminal ou através da tecla de atalho CONTROL

+ C.

Agora eu vou acessar a URL através do meu browser favorito, o Chrome:

```
localhost:3000
```

Nada acontece e o browser (nome mais bonitinho para navegador) fica pensando, pensando e com a mensagem lá na barra de status: *aguardando localhost....* Qual a razão disso? Será que nosso servidor está surdo e não sabe que estamos acessando-o?

Podemos verificar essa possível surdez através da função **listen**. Esta função além de receber a porta que queremos que nosso servidor escute, também recebe como segundo parâmetro um trecho de código que será chamado assim que nosso servidor estiver de pé e ciente do mundo ao seu redor.

Em Javascript, sabemos que podemos isolar um conjunto de instruções para executarmos depois através de uma função:

```
// alurapic/sever.js

var http = require('http');

http.createServer().listen(3000, function() {
    console.log('NEH');
});
```

Veja que no lado do servidor com o Node.js, temos também o objeto `console` assim como temos no navegador, inclusive ele tem as mesmas funções. A diferença, claro, é que exibiremos a mensagem no terminal e não no navegador, pois nosso código é um código de backend. Não custa nada dizer que não existe a função `alert` no Node.js, pois não existe tela de navegador para exibir a caixa de texto, né.

Muito bem, vou parar o servidor e executá-lo novamente, aliás, isso é necessário sempre que você alterar um arquivo do nosso servidor.

```
node server
```

E agora, abrir a URL novamente do nosso servidor:

```
localhost:3000
```

Olhando no terminal, recebemos a mensagem "NEH". Que lindo, nosso bebê está acordado e nos ouvindo! Porém, nosso browser ainda fica aguardando uma resposta. Essa criança não fala nada, só geme, estou preocupado.

Nosso bebê não fala... É mudo?

Queremos que nosso servidorzinho, coitadinho, dê uma resposta toda vez que alguém falar com ele. Não precisa ser uma resposta complexa, mas tem que ser apenas para aquela pessoa que está interagindo com ele. Para isso, precisamos passar uma função como parâmetro para a função `http.createServer` . Essa função será chamada pelo servidor toda vez que alguém entrar em contato com ele através do endereço (URL) `localhost:3000` :

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

http.createServer(function() {
    console.log('PAPAI');
})
.listen(3000, function() {
    console.log('NEH');
});
```

Vou parar o servidor e rodar novamente, para em seguida abrir novamente no meu browser o endereço. Aham! Dessa vez nosso servidorzinho responde com "PAPAI" no console do servidor, mas nosso browser continua esperando uma resposta.. Mas eu já não dou uma reposta com o `console.log` ? Não, não dou. O `console.log` apenas imprime no console do servidor um texto e não envia uma resposta para quem o acessou, razão pela qual nosso navegador fica eternamente esperando uma resposta.

Vou te contar um segredo: quem chama a função que passamos para o `createServer` ? Nosso servidor, certo? Chama quando? Toda vez que alguém tenta acessa-lo. Se tivermos 500 usuários acessando nosso servidor, essa função será chamada 500 vezes e assim por diante. Quando a função é chamada, o servidor sempre passará dois objetos como parâmetros para a função: um que representa o fluxo de requisição e outro o de resposta. Através do primeiro temos acesso a qualquer informação enviada para o server e através do segundo podemos enviar uma resposta para quem o acessou. Perfeito:

```
// alurapic/server.js

var http = require('http');

http.createServer(function(req, res) {
  res.end('PAPAI');
})
.listen(3000, function() {
    console.log('NEH');
});
```

E agora? Quando reiniciamos o servidor e acessamos a URL, recebemos um "PAPAI". Como resposta! Agora sim! O objeto que encarna a requisição feita possui a função `end` . Ela permite escrevermos uma mensagem para quem acessou nosso servidor e automaticamente a envia.

Diga olá para o mundo, queridinho da mamãe!

Mas se fosse a mamãe que tivesse falado com ele? Você concorda que esse texto teria que variar? Que tal se, quando acessarmos o servidor, nós enviarmos para ele uma informação dizendo que sou o papai ou a mãe e ele devolvesse como resposta papai ou mamãe, respectivamente?

No mundo HTTP (Ah , inclusive temos um treinamento do protocolo HTTP se isso lhe interessar), podemos passar parâmetro para o server da seguinte maneira:

```
http://localhost:3000?parente=MAMÃE
```

No exemplo acima, estamos enviando o parâmetro `parente` que possui como valor o texto `MAMÃE` . Será que funciona? Bem, não funciona, a resposta continua sendo `PAPAI` . E agora? Da mesma maneira que interagimos com o fluxo de resposta,

podemos interagir com o fluxo de requisição e obter a informação enviada através do navegador:

```
// alurapic/server.js

var http = require('http');

http.createServer(function(req, res) {
    var parente = req.url;
    res.end(parente);
})
.listen(3000, function() {
    console.log('NEH');
});
```

Vamos reiniciar e acessar a URL `http://localhost:3000?parente=MAMÃE`

Opa! Funciona, mas recebemos como resposta `?parente=MAMÃE`, quando na verdade só queríamos receber como resposta `MAMÃE`. E agora? Precisamos extrair essa informação que nos interessa, isto é, precisamos analisar `req.url` e convertê-la no formato que desejamos. O termo técnico desse procedimento é **parser**, ou seja, queremos realizar um *parser* de `req.url`:

A boa notícia é que no mundo Node.js, todas aquelas funções básicas de `String` também estão disponíveis, demonstrando mais uma vez que nosso conhecimento de Javascript é aproveitado no server:

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

http.createServer(function(req, res) {
    var parente = req.url.substr(req.url.indexOf('=') + 1);
    res.end(parente);
})
.listen(3000, function() {
    console.log('NEH');
});
```

Agora sim! Nossa resposta está no formato que esperamos.

Só tomando um café "Express" para ficar acordado

Meu bebê escuta e fala, mas ele precisa fazer muito mais do que isso para se tornar um homenzarrão pronto para ter atitude de server. Nosso server tem que ser capaz de servir para o navegador todo o conteúdo estático da pasta `alurapic/public`. Isso significa que se digitarmos `http://localhost:3000/index.html` ele deve nos devolver a página `index.html`. Mas você deve lembrar que temos arquivos de estilos e scripts que também são baixados quando abrimos nossa página. E agora? Um esboço de solução seria:

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

http.createServer(function(req, res) {
    if(req.url.startsWith('/index.html')) {
        console.log('envia página');
    }
    if(req.url.startsWith('/estilos.css')) {
        console.log('envia css');
```

```
    }
})
.listen(3000, function() {
    console.log('NEH');
});
```

Eu não preciso meditar muito para ver que meu servidor começará a ficar abarrotado de `if`'s, sem falar que nem entrei no assunto de manipulação de arquivos no sistema de arquivos com o Node.js. Pois é, até que nosso servidor amadureça, terei que tomar muito café para ficar acordado noite e dia cuidando da nossa criança até que ela cresça. Ah, por falar em café, o que eu mais gosto é o tipo **expresso**. Vou até realizar uma pesquisa no Google sobre esse tipo de café, quem sabe fazer um estoque para os dias que estão por vir.

Espere um pouco, como resultado de pesquisa apareceu **Express - Node.js web application framework**! Não podia ser melhor, e realmente, existe um framework Web para Node.js que pode tornar nosso bebê um adulto em questões de minutos. O Express (http://expressjs.com) é um framework web compacto e poderoso que pode nos poupar muito tempo resolvendo problemas como os que eu acabei de listar.

Solicitando o Express sem sair de casa

Express é um módulo como qualquer outro do Node.js que usamos, com a diferença que não vem como padrão na instalação do Node.js. A boa notícia é que podemos baixá-lo através do nosso terminal. O Node.js já vem por padrão com seu gerenciador de pacotes chamado **npm**, também executado através do terminal. Este gerenciador é capaz de acessar seu grande repositório online com zilhões de projetos open source e baixar os módulos de que precisamos.

Está ansioso para baixar o Express através do npm? Tenho certeza que sim, mas primeiro, precisamos criar uma "caderneta" onde tomaremos nota de todos os módulos utilizados pela nossa aplicação. Se um dia precisarmos descobrir rapidamente quais módulos utilizar, podemos olhar essa caderneta.

O uso da caderneta também é interessante, porque se um dia apagarmos todos os módulos do nosso sistema, poderemos baixá-los novamente porque temos todos anotados. No mundo Node.js, essa caderneta se chama **package.json**. Ela não guarda apenas os módulos baixados do nosso sistema, ela também guarda o nome do autor, nome do projeto, versão e outras informações.

Mas onde está esse arquivo? Ele ainda não existe e precisamos criá-lo. Certifique de estar dentro da pasta **alurapic** e em seguida execute o comando:

```
npm init
```

Uma série de perguntas será feita, mas não se preocupe agora, queremos apenas criar o arquivo e para isso podemos dar ENTER para todas elas até que o assistente termine. No final, teremos nosso arquivo `package.json`. Ele terá essa estrutura:

```
{
  "name": "alurapic",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
```

```
  "author": "",
  "license": "ISC"
}
```

Se você não reconhece essa estrutura, talvez seja necessário verificar os pré-requisitos do treinamento. É uma estrutura amplamente difundida entre programadores Javascript, a estrutura JSON. Veja que o Node.js não inventou nada novo, pegou carona em uma estrutura já existente.

Ainda sobre o package.json, veja que há uma chamada de nome `"start"` e seu valor é `"server.js"`. O que isso significa? Quando rodamos o `npm init` o assistente encontrou um único arquivo no mesmo diretório em que foi chamado e por isso ele considerou aquele arquivo o ponto de entrada da nossa aplicação. A vantagem disso, é que no lugar de rodarmos o sevidor com o comando `node server`, podemos usar a partir de agora:

```
npm start
```

Por debaixo dos panos, o que o npm executará será o comando `node server`. Vamos combinar de usarmos no `npm start` até o fim do treinamento!

Agora que temos nosso arquivo, podemos instalar o express através do `npm` com o comando:

```
npm install express@4.13.3 --save
```

Caso o acesso a internet seja realizado através de um proxy, será necessário realizar a configuração do mesmo, conforme exemplo abaixo:

```
npm config set proxy http://proxy.company.com:8080
```

Durante algum tempo, seu terminal ficará trabalhando e baixando o Express e todas as suas dependências. Quais dependências? Não faço ideia, deixe o `npm` resolvê-las para nós! Estou instalando a versão 4.13.3, sendo importante que você utilize a mesma versão, pois essa eu já testei e não há nenhum bug que interfira em nosso projeto. Aliás, quando for pedir ajuda no fórum eu sempre perguntarei se você está usando a mesma versão da biblioteca que utilizei no treinamento. Esta é a versão mais atual do Express na data de criação deste treinamento.

Quando tudo tiver sido baixado, será criada uma pasta chamada `alurapic/node_modues/express`. Não se preocupe, por padrão, todas os módulos baixados pelo `npm` ficam dentro do diretório no qual você executou o `npm`, mas dentro da pasta `node_modules`. Aliás, esses módulos são dependências do nosso projeto que não funcionaria sem eles. Além disso, será criada uma pasta com o nome do pacote que baixamos, em nosso caso, `express`. Perfeito, já temos o Express baixado. Agora só precisamos configurá-lo.

Dividindo para conquistar

Precisamos configurar nosso Express e todos os arquivos de configuração do nosso sistema ficarão dentro do diretório `alurapic/config`. Eu peço que você não mude o nome dos diretórios durante o treinamento porque não é incomum dúvidas no fórum do aluno que trocou os nomes dos arquivos e diretórios e esqueceu de trocar em alguns lugares, combinado? **Vou criar o diretório** `alurapic/config`.

Agora, vamos criar o arquivo `alurapic/config/express.js`, nosso módulo que configurará o express.

```
// alurapic/config

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

Nossa primeira linha faz o requerimento, isto é, a importação do módulo `express`, que armazenamos na variável `express`. Uma dica: passamos para a função `require` o nome do módulo que baixamos através do `npm`. Como o módulo do Express se chama "express", usamos este nome.

Muito bem, importar o Express ainda não é suficiente, precisamos de uma instância dele para podermos trabalhar.

```
// alurapic/config

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

Agora, nossa variável `app` que guarda uma instância do Express. É claro que ela precisa ser configurada, mas já já faremos isso. O que eu quero primeiro fazer aqui é brincar de lego, isto é, mostrar como o módulo `alurapic/config/expres.js` se encaixa com o módulo `alurapic/server.js` que também criamos. Depois de encaixarmos um no outro, podemos voltar e configurar nossa instância do Express.

Vou abrir para edição `alurapic/server.js` e importar nosso módulo de configuração. Já sabemos que a importação de um módulo é feita através da função `require`:

```
//alurapic/server.js

var http = require('http');
var app = require('./config/express'); // novidade aqui!

http.createServer(function(req, res) {
    if(req.url.startsWith('/index')) {
        console.log('envia');
    }
})
.listen(3000, function() {
    console.log('NEH');
});
```

Como nosso módulo não faz parte de `node_modules`, foi necessário colocar o `./` no caminho do módulo, caso contrário não o encontraríamos. Em seguida, no lugar de passarmos uma função que recebe o fluxo de requisição e resposta, passamos como parâmetro para o `createServer` nossa instância do Express!

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

http.createServer(app)
.listen(3000, function() {
    console.log('NEH');
});
```

Consegue perceber? Express nada mais é do que um módulo que aplicará uma pilha de filtros para lidar com a requisição e resposta recebido pelo nosso servidor. Na verdade, o termo correto não é "pilha de filtro", mas sim **pilha de middlewares**.

Middlewares são funções que lidam com requisições. Uma pilha de middlewares pode ser aplicada em uma mesma requisição para se atingir diversas finalidades (segurança, logging, auditoria etc.). Cada middleware passará o controle para o próximo até que todos sejam aplicados.

Perfeito, agora você sabe que Express é um módulo que pode aplicar uma série de funções para atingirmos diversas finalidades. Será que já podemos subir nossa aplicação? Vamos testar?

```
npm start
```

Ops, recebemos um erro:

```
events.js:197
    throw new TypeError('listener must be a function');
    ^

TypeError: listener must be a function
```

Isso acontece, porque não passamos uma função para o `http.createServer`. Mas como não, se o Express é justamente aquela função com papal de middleware, que pode adicionar outros middlewares em sua configuração?

Se imprimirmos a variável `app`, teremos uma surpresa:

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

console.log(app); // imprimindo o retorno do módulo ./config/express

http.createServer(app)
.listen(3000, function() {
    console.log('NEH');
});
```

No console, antes da mensagem de erro, é exibido um objeto javascript sem qualquer propriedade, um `{ }`. Isso acontece porque a chamada da função `require` para nosso módulo `./config/express` não retornou nossa instância do Express? Por quê?

Você lembra quando eu disse que um módulo é uma unidade de código confinada? Pois é, todo código do nosso módulo `config/express` não é enxergado. É como se fosse uma peça de lego sem "dente". Precisamos exportar os dentes dessa peça e isso o Node.js não faz automaticamente para nós. Precisamos **exportar** a funcionalidade que desejamos compartilhar com quem chamar o módulo. Sendo assim, vamos alterar nosso `config/express.js`:

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

// configurações do express que ainda faremos

module.exports = app; // exportando nossa instância do Express
```

Será que funciona agora?

Funciona, aliás não é mais impresso o `{ }`, mas uma função construtora cheia de parafernalha, que é o Express. Não precisamos mais desse `console.log(app)`. Vamos removê-lo, inclusive já podemos substituir a mensagem de quando nosso servidor está de pé por uma mais séria:

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

http.createServer(app)
.listen(3000, function() {
    console.log('Servidor iniciado');
});
```

Excelente! Agora o Express será capaz de lidar com as requisições e respostas do nosso servidor. Porém, ele nada faz. Que tal resolvermos aquele problema de compartilharmos a pasta `public` para que seja acessada através do navegador? É lá que moram nossos arquivos do projeto Angular.

Para fazermos isso, precisamos adicionar nosso primeiro middleware ao Express. Preparado? Queremos que o Express **use** o middleware, é por isso que associamos um middleware com o Express através da função **use**. Vamos alterar `config/express`:

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

// nosso primeiro middleware
app.use(express.static('./public'));

module.exports = app;
```

A função `app.use` recebe como parâmetro `express.static('./public')`, o middleware de arquivos estáticos. Este middleware recebe como parâmetro o nome do diretório que desejamos compatilhar, em nosso caso estamos compartilhando `alurapic/public`, mas indicamos isso através de `./public`.

Agora, vamos reiniciar o servidor e acessar:

```
localhost:3000/index.html
```

Agora sim! Inclusive podemos acessar a URL:

```
localhost:3000
```

Quando não passamos nada, o padrão é devolver o `index.html`! Conseguimos exibir a página principal que construímos em Angular do outro treinamento, porém ela não exibe informação alguma. Se abrirmos o console do Chrome, veremos mensagens de erro, inclusive uma com o código 404 (não encontrado). Isso acontece porque nosso servidor não disponibilizou a URL `http://localhost:3000/v1/fotos`, aquela que no treinamento de Angular fornecia uma lista de fotos para a aplicação Angular.

Aprendemos que essas URL's são endpoints REST e sua criação em nosso servidor é assunto do próximo capítulo.