Your First Node App: Make A Twitter Bot

Emily Aviva Kapor-Mater

Published with GitBook

Table of Contents

Introduction

Your First Node App: Build A Twitter Bot

Introduction

Have you ever wondered what the big deal is with Node?

This book grew out of a tutorial I wrote after developing some of my own Twitter bots.

About This Book

This book was written by Emily Aviva Kapor-Mater (website, github, twitter). It was last substantively updated 11 December 2015.

Getting Set Up

Chapter 1: Getting Set Up

Tools You Will Need

In this tutorial, I assume you know how to use a text editor like Atom, and are at least somewhat comfortable with the terminal command line.

If you haven't got it already, you should install Node.

You should have an account (free or otherwise) at Heroku, and you should download and install the Heroku toolbelt.

A GitHub account is a very good idea so you can store and show off your code, but is not required. You do, however, need to have Git installed on your computer. (It comes by default with the Mac and most Linux versions.)

Preliminary setup

We are writing our bot in Node, which is a runtime for JavaScript. We will be doing version tracking with Git. We need to initialize our new application by making an empty directory, making an empty Git repository, and then using `npm` (the Node package manager) to start a new blank app.

```
mkdir my-first-node-app
cd my-first-node-app
git init
npm init
```

For the moment, it's okay to just hit "enter" if you don't know what to write when `npm init` prompts you for input. We'll come back to it later. For now, we'll end up with a file in the root level of our project directory called `package.json`, which is a basic configuration file for our new Node application, and a new directory called `node_modules`, which will contain **dependencies**: third-party packages that we will use, basically, to avoid having to write everything ourselves.

We'll also want to create a file called `.gitignore`, which tells Git not to track everything listed in it. The only thing here we don't want to track is the `node_modules` directory that will store our app's dependencies. We can create the `.gitignore` file from our text editor, or we can create and write it from the command line:

```
echo "node_modules" >> .gitignore
```

Making a Tweet Generator

Chapter 2: Making a Tweet Generator

Introducing libraries

Before we can actually tweet anything, we need to have a tweet generator: otherwise, we'd just be tweeting lots of blank tweets. The generator itself is actually not going to be part of the bot app itself, but instead it will constitute a separate **library**: a script with reusable code.

Let's set up a directory called `lib` in our project directory and make a file for our generator inside it. We'll also need another library of words and phrases for our generator, so let's make that too.

```
mkdir lib
touch lib/generate.js lib/dictionary.js
```

The most important thing our generator program will do is return a string, which we can then use as the body of a tweet. While we're setting it up, let's also have it pull in our dictionary, even though there's nothing in it yet.

```
// generate.js
'use strict';

var dictionary = require('./dictionary');

function generate() {
  return 'Hello, world!';
}

module.exports = generate;
```

The last line tells Node that when this file is imported from another program with the `require()` command, we want it to make the `generate()` function available. (We don't put `()` after the function name here, because we want to export the function itself; not the *execution* of that function: a subtle but crucial distinction.) So, as you might be guessing, we will need to have a similar `module.exports` statement in our dictionary library as well:

```
// dictionary.js
'use strict';

var dictionary = {};

module.exports = dictionary;
```

Generating a random string

Right now, our dictionary is simply an empty object. Let's fill it with some words. But which words? I know! Let's fill it with some words that will help us build up a bot to generate pretentious food truck names.

```
var dictionary = {
  foods: ['sandwiches', 'wraps', 'pies', 'bowls', 'burgers'],
  ingredients: ['quinoa', 'chanterelle', 'oyster', 'leek', 'algae'],
  descriptions: ['burnt', 'toasted', 'sizzling', 'sprouted', 'gentri
  truckTypes: ['truck', 'cart', 'vehicle', 'conveyance', 'brougham',
}
```

To generate a food truck name, we need a function that will pick a random element from each of these arrays and string them together properly. Basically, we want to be able to say something like `dictionary.foods[n]` where the number is a random integer between 0 and the length of the `foods` array.

Randomness in JavaScript is a little tricky: there isn't a built-in function that will let us pick a random element from a list. We could build up a function to do this, but it's already been done for us in a wonderful library called Lodash, which is traditionally abbreviated to _ in JavaScript.

To install Lodash, on the terminal command line, we do:

```
npm install --save lodash
```

The `--save` tells `npm` that we want to install Lodash as a dependency for this project: in other words, our bot won't run unless Lodash is available. If we open up the `package.json` file now, we'll see that there's a section at the bottom called `"dependencies"`, and one of the items listed will be `"lodash"`, followed by a minimum version number. Node and `npm` will track our dependencies for us, so we don't need to worry about installing it once it's listed in `package.json`.

Let's go back to our `generator.js` and include some Lodash to capitalize our randomly selected words. The Lodash function `_.sample()` takes a collection and returns one (or more) random elements from it. Let's edit our `generate()` function to one item from each array in the `dictionary` object, capitalize it, and concatenate them all together.

```
var _ = require('lodash');

function generate() {
  var food = _.capitalize(_.sample(dictionary.foods));
  var ingredient = _.capitalize(_.sample(dictionary.ingredients));
  var description = _.capitalize(_.sample(dictionary.descriptions));
  var truckType = _.capitalize(_.sample(dictionary.truckTypes));
  return description + ' ' + ingredient + ' ' + food + ' ' + truckTyp
}
```

If we wanted, we could verify our function by adding a command to the end of `generate.js` to log our output to the console:

```
console.log(generate());
```

Then, we can run `node generate.js` from the terminal command line:

```
$ node generate.js
Burnt Oyster Pies Vehicle
$ node generate.js
Burnt Quinoa Burgers Brougham
$ node generate.js
Toasted Leek Sandwiches Chariot
```

Using `console.log()` as a debugging tool is a good practice. It would, however, be a good idea to remove the `console.log()` from the final production-ready script, because otherwise it'll just clutter up our console.

Fine-tuning the generator

Our output is looking pretty good so far, but it's not quite tweet-able yet. We'd like a couple of different types of output, so we don't repeat ourselves too often. What if we want a few different kinds of truck names, following certain patterns? Let's make four patterns:

- The Burger Truck
- Burnt Oyster Vehicle
- The Toasted Leek Sandwich
- The Sizzling Chanterelle Bowl Conveyance

Given these patterns, and assuming we want them to occur with equal frequency, three-quarters of the time we want "The" to be prefixed to the generated string, a different three-quarters of the time we want the truck type to be at the end, and so forth. There are multiple ways to write something like this, but let's do it this way for now: we'll use Lodash's `_.random()` function to generate a random number between 1 and 4 (which will look like `_.random(1, 4)`), and depending on the result have our `generate()` function follow a different output pattern.

Finally, we'll have to make our `generate()` function importable from another JavaScript file in Node. To do this, similar to how we did with the `dictionary.js` script, we'll add the line `module.exports = generate` to the end of the `generate.js` script.

Our `generate.js` file now should look something like this:

```
// generate.js
'use strict';

var _ = require('lodash');
var dictionary = require('./dictionary');

function generate() {
  var food = _.capitalize(_.sample(dictionary.foods));
  var ingredient = _.capitalize(_.sample(dictionary.ingredients));
  var description = _.capitalize(_.sample(dictionary.descriptions));
  var truckType = _.capitalize(_.sample(dictionary.truckTypes));

  var output = '';
  var randomNumber = _.random(1, 4);

  if (randomNumber === 1) {
    output = 'The ' + food + ' ' + truckType;
  } else if (randomNumber === 2) {
    output = description + ' ' + food + ' ' + truckType;
  } else if (randomNumber === 3) {
    output = 'The ' + description + ' ' + food;
  } else {
    output = 'The ' + description + ' ' + food + ' ' + truckType;
  }
```

```
  return output;
}

module.exports = generate;
```

Using the Twitter API

Chapter 3: Using the Twitter API

Once we've created a new Twitter account for our bot to tweet to (and linked it to a phone number—Twitter requires this for new apps), we have to register the app that will be sending it tweets to post with Twitter's developer service. We log in with our new Twitter account, go to Twitter Apps, and click "Create New App". We give our app a name, a description, and a placeholder website since we haven't got our app up and running yet. We can ignore "callback URL", too. We agree to the developer terms and conditions, and presto, we have a new Twitter app.

We will need to make note of our app's keys and access tokens. When we're looking at our app's description, we can click on the "Keys and Access Tokens" tab to get our app's consumer key and consumer key secret.

There are four strings of gibberish we need to send to the Twitter API when we're making tweets. Two are available on this page: "Consumer Key (API Key)" and "Consumer Secret (API Secret)". The other two will be generated when we click on the "Create Access Token" button at the bottom of the screen: we'll get our "Access Token" and our "Access Token Secret". Verify that the access level is "Read and write", and we're good to go.

There's no need to write down these gibberish strings now, or, in fact, ever. In fact, writing them down or putting them into a file can be extremely *unsecure*, since that puts them out there in the open for anyone to read. Later on, we'll put them into our app as environment variables.

Writing a Node Server

Chapter 4: Writing a Node Server

Setup

While we could write the application that controls our bot in vanilla Node, we're going to use the Express framework for building the guts of our bot, since it gives us easy access to things like REST routing, which could come in handy if we ever want to extend our bot.

To save Express and Twitter as dependencies in our bot, we run:

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

We can check our `package.json` under `"dependencies"` and sure enough, `"express"` and `"twitter"` are now listed, along with `"lodash"`. This means that when we install our bot on Heroku, we will only need to copy over the `package.json` file and not the `node_modules` directory that we just created. In fact, let's tell Git to ignore `node_modules`, since we can install them at will with `npm` and we don't need our application to be carrying around all their weight all the time.

```
echo "node_modules" >> .gitignore
```

This creates an "invisible" file called `.gitignore` (or adds to it if it already exists) and adds a line telling Git to ignore the `node_modules` directory when we're pushing our repository to GitHub.

A simple server

Let's create a new file at our project root called `server.js`. This file will be the "entry point" for our application.

```
// server.js
'use strict';

var express = require('express');
var app = express();
var port = 3000;

app.get('/', function(req, res) {
  res.send('Congratulations, you sent a GET request!');
  console.log('Received a GET request and sent a response');
});

app.listen(port, function() {
  console.log('App now listening on port', port);
});
```

Let's go through this server piece by piece. First, we import the Express module, and then instantiate a new app with Express. We then declare the port that our app will be listening on, which we will be

setting dynamically later on, but for now we're just hard-coding it to 3000.

We then set an Express route so that anyone who goes to our app's URL will get some data back. Here, we're using the HTTP `GET` verb. The `app.get()` function takes as its second argument a callback function with (at least) two parameters one to represent the request object that was sent to the server, and the other to represent the response object that we will send back. In this case, the only thing we are sending back via our request object is a string confirming that the GET request was sent, received, and processed correctly. We then have our app print a message to our console telling us that the GET request was received and responded to.

Finally, we tell our app to listen for connections on the port we defined earlier, and execute a callback function when the app is up and running. Our callback function simply prints a message to our console confirming that the app is up and running and listening on the port we gave it.

Let's run this file and test it out. In our terminal, we execute this file with the `node` command:

```
node server.js
```

If our file is written correctly and there are no errors, Node will respond with the line

```
App now listening on port 3000
```

followed by a blank line, since our app is still running and listening for connections. We can test our app by opening up a web browser and going to `http://localhost:3000` (which sends a GET request to our local server on port 3000). If all goes well, the browser should display the "congratulations" message, and our app should print a new line stating that a GET request was received and responded to.

We've just constructed the of a complete Node server. It could be extended in potentially infinite directions from here with more routing and functionality. But let's continue in a different direction, and extend our app so that it is not only listening for connections, but sending its own.

Using the Twitter API

In order to make our app talk to Twitter, we need to use the Twitter package we installed earlier. Once we've added it to our server program, we will have access to an object constructor function called `Twitter()`, which has a whole lot of built-in goodies to make our job of posting tweets pretty simple.

We add the following to the beginning of our `server.js`:

```
var Twitter = require('twitter');
```

Once we've done that, we can create a new `Twitter` object from an object literal. It needs to have four properties, as follows:

```
var tweet = new Twitter({
  consumer_key: '',
  consumer_secret: '',
  access_token_key: '',
  access_token_secret: ''
});
```

Important note: At the moment, it is okay to replace these four properties with the values taken from

our Twitter app, but this is something that we **absolutely must remove** before we put our app on GitHub or share our code on the web at all. There are bots that scrape secret keys and similar data, so we want to take them out. We'll go over what to replace it with later on.

The `tweet` object inherits a method from the `Twitter()` constructor called `post()`. We'll use the `generate()` function we defined earlier to provide the text for our status. First, we need to import our `generate()` function from `generate.js`:

```
var generate = require('./lib/generate');
```

Since we set `module.exports` equal to the `generate()` function in `generate.js`, when we require `generate.js` we get back that exported function.

Now we can define a function in our server to post a tweet.

```
function makeTweet() {
  var content = generate();
  tweet.post('statuses/update', {status: content}, function(error, tw
    if (error) throw error;
    console.log('Posted tweet: \"' + content + '\"');
  });
}
```

The `post()` method on the `tweet` object takes three arguments:

1. The REST endpoint from the Twitter API (in this case, `statuses/update`, since we are creating a new tweet)
2. An object with the key `status` corresponding to the content of the tweet that we generated with our `generate()` function
3. A callback function that handles what happens when our `POST` request is made. If there is an error, we report it; otherwise, we write out a console message telling us that the tweet posted successfully. (We're going to keep the `tweet` and `res` (response) parameters in our callback function even though we're not explicitly using them now, because it will make it easier to do so in the future if we wish.)

In order to actually make a tweet, we need to call our `makeTweet()` function. But we don't want to make only one tweet; we want to make multiple tweets at certain intervals. We could hard-code this interval, or, even better, we could put it into a variable so we can change it if we wish. JavaScript keeps time in milliseconds, so if we want our app to post a new tweet, say, every thirty minutes, we would write that value as 1000 *60* 30 = 1,800,000 milliseconds.

We first call `makeTweet()` to make our first tweet, and then use the `setInterval()` function to tell our application to do it again every `tweetInterval` milliseconds.

```
var tweetInterval = 1800000;

setInterval(makeTweet, tweetInterval);
```

When we write that in `server.js`, the file now should look something like this:

```
// server.js
'use strict';

var Twitter = require('twitter');
```

```
var express = require('express');
var generate = require('./lib/generate');
var app = express();
var port = 3000;
var tweetInterval = 1800000;

var tweet = new Twitter({
  consumer_key: '',       // replace with
  consumer_secret: '',    // values from
  access_token_key: '',   // our Twitter
  access_token_secret: '' // app data
});

function makeTweet() {
  var content = generate();
  tweet.post('statuses/update', {status: content}, function(error, tw
    if (error) throw error;
    console.log('Posted tweet: \"' + content + '\"');
  });
}

app.get('/', function(req, res) {
  res.send('Congratulations, you sent a GET request!');
  console.log('Received a GET request and sent a response');
});

app.listen(port, function() {
  console.log('App now listening on port', port);
});

makeTweet();
setInterval(makeTweet, tweetInterval);
```

Now, let's make some magic happen. Let's run our app and watch it post a tweet.

```
node server.js
```

If everything's been written correctly, and there are no errors, we should have a randomly generated tweet appear in our timeline! If we leave our app running for the length of our interval, we'll see another randomly generated tweet appear.

Deploying Our App

Deploying Our App

So far, this is pretty good. But in order to run it constantly, we'd have to keep our own computer online permanently and never close this server. Fortunately, there are many services that will host our application and keep it running. One such "platform-as-a-service" is Heroku. As of this writing, they'll even host some low-impact apps for free.

I assume you've already set up your Heroku account and installed the Heroku toolbelt, and that you've used `heroku login` to log in to your Heroku account on the command line.

Preparing for Heroku

Heroku pulls our app code from Git. This means that we need to clean our code of private secrets before we commit it and push it to GitHub.

Let's change our `tweet` object to scrub out the secrets:

```
var tweet = new Twitter({
  consumer_key: process.env.TWITTER_CONSUMER_KEY,
  consumer_secret: process.env.TWITTER_CONSUMER_SECRET,
  access_token_key: process.env.TWITTER_ACCESS_TOKEN_KEY,
  access_token_secret: process.env.TWITTER_ACCESS_TOKEN_SECRET
});
```

The `process` object is particular to Node: it refers to a whole slew of data, such as how the app was invoked, what paths it's looking for modules in, and so forth. Here, we're concerned with its `env` property, which stores **environment variables**. An environment variable is a piece of data that's set on the host computer, and is available to programs that run on that computer. We'll actually create these environment variables later; for now, we're just setting it up so that our code knows to look for them.

Let's also change our other "global" variables in `server.js`.

```
var port = process.env.PORT || 3000;
var tweetInterval = process.env.TWEET_INTERVAL || 1800000;
```

Here, we're asking our computer if it has an environment variable called `PORT`, and another one called `TWEET_INTERVAL`, and if those exist, assign their values to our application's `port` and `tweetInterval` variables. If those don't exist, we use the logical NOT operator `||` to set the values to `3000` and `1800000`, respectively.

If you're running the `bash` shell on your command line, you can set environment variables with the `export` command, e.g.:

```
export PORT=3333
```

In the `fish` shell, use:

```
set -x PORT 3333
```

We can use the fact that we're setting these values to be variables pulled from the process environment to change them as we wish: if we want our app to tweet more or less frequently, all we have to do is change the value of an environment variable, rather than change a line that's baked into our code. And it's far more secure to do things this way than to leave your secret keys hard-coded into a document that's publicly available on GitHub. Don't do that. I mean it.

Creating a Heroku application

In the root level of our application, we'll run a command to initialize our bot as a Heroku application:

```
heroku create <name>
```

Replace `<name>` with the name you want your app to possess, i.e. the part that comes in its URL before `.herokuapp.com`. If you don't assign a name, Heroku will automatically assign your app a bad name like "fathom-bunches-2187". In your Heroku web panel, you can reset the app's name to something better, like "my-first-twitter-bot".

The Heroku tools will work by looking for a remote on our Git repository called `heroku`, which is automatically added by the `heroku create` command. You can verify this by running `git remote -v` from your terminal; you should now called `heroku`. When you `git push` to the `master` branch on the `heroku` repository, you are telling Heroku "update the current code base "

Keeping our server alive

Heroku turns off free apps if they've been idle for a certain amount of time. We need to make sure that our app receives some kind of HTTP request every so often, so that Heroku will keep its process alive. Fortunately, we already wrote an Express route to handle getting a request, but we need a little more work in order to get our app to ping itself every so often.

To send an HTTP request from our app, we'll need to use the `http` package. This is a Node built-in package, so we don't need to explicitly `npm install` it. We'll import it into our `server.js` up near the beginning, where we're loading our other packages. Then, we can tell our server to send a `GET` request to the route at `/`, which we already defined with Express, every ten minutes:

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

function keepAlive() {
  try {
    http.get('http://[put the name of your app on Heroku here].heroku
    console.log('GET request sent; kept alive.');
  } catch(e) {
    console.log(e);
  }
}

setInterval(keepAlive, 600000);
```

What this means is that, unless there's an error (which we log to the console), send a `GET` request to our server at the `/` (root) route every 600,000 milliseconds = 10 minutes, and then print a console message stating that it happened. (Strictly speaking, the `console.log()` statement isn't necessary here, since we'll get a log that a `GET` request was received, but let's make our app as explicit as

possible about what's happening.)

Commit these changes to Git, and then `git push heroku master` once more. Heroku will rebuild the app with its new dependency and code; then the app will restart, and now it will also automatically keep itself alive.

Setting environment variables

We're expecting our app to get quite a few values from environment variables. Setting environment variables on a Heroku app is pretty simple. It can be done from the Heroku web dashboard, or from the command line. Let's do it the second way.

The Heroku toolbelt provides the command `heroku config:set` for setting an environment variable. Let's say we want our app to tweet once per hour:

```
heroku config:set TWEET_INTERVAL=3600000
```

Using this procedure, we'll set every environment variable we'll need. (We won't need an explicit `PORT` because Heroku automatically assigns that.) We can get the values of the four Twitter API keys and tokens from the Twitter apps page under the "Keys and Access Tokens" tab.

```
heroku config:set TWITTER_CONSUMER_KEY=
heroku config:set TWITTER_CONSUMER_SECRET=
heroku config:set TWITTER_ACCESS_TOKEN_KEY=
heroku config:set TWITTER_ACCESS_TOKEN_SECRET=
```

Of course, put the appropriate line of gibberish after the equals sign. If you want to check what you've written, you can use the `heroku config` command (without `:set`) to get the current value of your variables, or `heroku config:get` to get an individual variable's value. If you've typed something wrong, you can unset a variable by using `heroku config:unset`.

Starting up the server

We need to upload our app's code to Heroku and then, basically, turn it on. For a Node app, Heroku builds its codebase based on what's inside your `package.json` file. The most important value in `package.json` as far as Node is concerned is `"main"`: this is the name of the file that Node should start your application running with, sometimes referred to as the app's *entry point*.

from whatever is in the `master` branch of Git's `heroku` remote repository. So, we need to add all our files to Git, commit them, and push them to where we want them to go.

```
git add .
git commit -m "initial push to Heroku"
git push heroku master
```

Heroku will churn away for a while, setting up our app, installing all the dependencies listed in `package.json`, and doing lots of other little tasks. Once it's complete, all we have to do is turn the key in the ignition.

A Heroku app runs in something called a "dyno", which is basically a running copy of your app. We will only need one dyno for our app. Our Heroku app won't actually start until we allocate a dyno to it, which we do from the command line as follows:

```
heroku ps:scale web=1
```

Heroku has two types of dynos: for "web" processes and for "worker" processes. Our app will use a "web" process, since it has an HTTP endpoint at which it will be pinging itself in order to keep itself alive. This command starts up one dyno to run our app.

Once the server is running, we should expect our bot to tweet immediately, and then again every `TWEET_INTERVAL` milliseconds (or the default value).

Summing Up

Summing Up

Congratulations! We've successfully built an app with Node, got it talking to Twitter, and running on Heroku. If you missed some of the code, you can find it in the GitHub repository, under `src/tutorial/`.

Our bot is pretty basic. It only has a few words in its dictionary. There are a whole bunch of places we could make improvements, particularly to the `generate()` function. The good news is that because our app is modular, in order to make changes we have only to change the file in which the data or code to be changed is located. When we redeploy our app to Heroku with `git push heroku master`, the app will automatically rebuild to the latest committed code base.

What next?

Some ideas on how to extend this bot:

- More patterns for food truck names
- Get more words from larger word databases (e.g. the Wordnik API)
- Respond to replies
- Favorite mentions

The code located in `src/production/`, which is used to power the example bot derived from this tutorial, has a couple of extra bells and whistles that you can use as a basis to imitate and extend. Don't be afraid to experiment!