Getting Started on Heroku with Node.js

Last updated 24 October 2019

Table of Contents

Introduction

This tutorial will have you deploying a Node.js app to Heroku in minutes.

Hang on for a few more minutes to learn how to get the most out of the Heroku platform.

The tutorial assumes that you have a free Heroku account (https://signup.heroku.com/signup/dc), and that you have Node.js and npm (https://nodejs.org/en/download/) installed locally.

Set up

The Heroku CLI requires **Git**, the popular version control system. If you don't already have Git installed, complete the following before proceeding:

- Git installation (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)
- First-time Git setup (https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup)

In this step you'll install the Heroku Command Line Interface (CLI). You use the CLI to manage and scale your applications, provision add-ons, view your application logs, and run your application locally.

Download and run the installer for your platform:

macOS

Download the installer (https://cli-assets.heroku.com/heroku.pkg)

Also available via Homebrew:

```
$ brew install heroku/brew/heroku
```

Windows

Download the appropriate installer for your Windows installation

64-bit installer (https://cli-assets.heroku.com/heroku-x64

32-bit installer (https://cli-assets.heroku.com/heroku-x86

Ubuntu 16+

Run the following from your terminal:

```
$ sudo snap install heroku --classic
```

Snap is available on other Linux OS's as well (https://snapcraft.io).

When installation completes, you can use the `heroku` command from your terminal.

Use the `heroku login` command to log in to the Heroku CLI:

```
$ heroku login
heroku: Press any key to open up the browser to login or q to exit
 ›   Warning: If browser does not open, visit
 ›   https://cli-auth.heroku.com/auth/browser/***
heroku: Waiting for login...
Logging in... done
Logged in as me@example.com
```

This command opens your web browser to the Heroku login page. If your browser is already logged in to Heroku, simply click the **`Log in`** button displayed on the page.

This authentication is required for both the `heroku` and `git` commands to work correctly.

If you're behind a firewall that requires use of a proxy to connect with external HTTP/HTTPS services, **you can set the `HTTP_PROXY` or `HTTPS_PROXY` environment variables (https://devcenter.heroku.com/articles/using-the-cli#using-an-http-proxy)** in your local development environment before running the `heroku` command.

Before you continue, check that you have the prerequisites installed properly. Type each command below and make sure it displays the version you have installed. (Your versions might be different from the example.) If no version is returned, go back to the introduction of this tutorial and install the prerequisites.

All of the following local setup will be required to complete the "Declare app dependencies" and subsequent steps.

This tutorial will work for any version of Node greater than 8 - check that it's there:

```
$ node --version
v12.13.0
```

`npm` is installed with Node, so check that it's there. If you don't have it, install a more recent version of Node:

```
$ npm --version
6.11.3
```

Now check that you have `git` installed. If not, install (https://git-scm.com/downloads) it and test again.

```
$ git --version
git version 2.17.0
```

Prepare the app

In this step, you will prepare a sample application that's ready to be deployed to Heroku.

If you are new to Heroku, it is recommended that you complete this tutorial using the Heroku-provided sample application.

However, if you have your own existing application that you want to deploy instead, see **this article (https://devcenter.heroku.com/articles/preparing-a-codebase-for-heroku-deployment)** to learn how to prepare it for Heroku deployment.

To clone a local version of the sample application that you can then deploy to Heroku, execute the following commands in your local command shell or terminal:

```
$ git clone https://github.com/heroku/node-js-getting-started.git
$ cd node-js-getting-started
```

You now have a functioning Git repository that contains a simple application as well as a `package.json` file, which is used by Node's dependency manager.

Deploy the app

In this step you will deploy the app to Heroku.

Create an app on Heroku, which prepares Heroku to receive your source code.

```
$ heroku create
Creating sharp-rain-871... done, stack is heroku-18
http://sharp-rain-871.herokuapp.com/ | https://git.heroku.com/sharp-rain-871.git
Git remote heroku added
```

When you create an app, a git remote (called `heroku`) is also created and associated with your local git repository.

Heroku generates a random name (in this case `sharp-rain-871`) for your app, or you can pass a parameter to specify your own app name.

Now deploy your code:

```
$ git push heroku master
Counting objects: 488, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (367/367), done.
Writing objects: 100% (488/488), 231.85 KiB | 115.92 MiB/s, done.
Total 488 (delta 86), reused 488 (delta 86)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Node.js app detected
remote:
remote: -----> Creating runtime environment
remote:
remote:        NPM_CONFIG_LOGLEVEL=error
remote:        NODE_VERBOSE=false
remote:        NODE_ENV=production
remote:        NODE_MODULES_CACHE=true
remote:
remote: -----> Installing binaries
remote:        engines.node (package.json):  12.x
remote:        engines.npm (package.json):   unspecified (use default)
remote:
remote:        Resolving node version 12.x...
remote:        Downloading and installing node 12.13.0...
remote:        Using default npm version: 6.12.0
      ....
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 19M
remote: -----> Launching...
remote:        Released v3
remote:        http://sharp-rain-871.herokuapp.com deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/nameless-savannah-4829.git
 * [new branch]      master -> master
```

The application is now deployed. Ensure that at least one instance of the app is running:

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

Now visit the app at the URL generated by its app name. As a handy shortcut, you can open the website as follows:

```
$ heroku open
```

View logs

Heroku treats logs as streams of time-ordered events aggregated from the output streams of all your app and Heroku components, providing a single channel for all of the events.

View information about your running app using one of the logging commands (https://devcenter.heroku.com/articles/logging), `heroku logs --tail`:

```
$ heroku logs --tail
2011-03-10T10:22:30-08:00 heroku[web.1]: State changed from created to starting
2011-03-10T10:22:32-08:00 heroku[web.1]: Running process with command: `node index.js`
2011-03-10T10:22:33-08:00 heroku[web.1]: Listening on 18320
2011-03-10T10:22:34-08:00 heroku[web.1]: State changed from starting to up
```

Visit your application in the browser again, and you'll see another log message generated.

Press `Control+C` to stop streaming the logs.

Define a Procfile

Use a Procfile (https://devcenter.heroku.com/articles/procfile), a text file in the root directory of your application, to explicitly declare what command should be executed to start your app.

The `Procfile` in the example app you deployed looks like this:

```
web: node index.js
```

This declares a single process type, `web`, and the command needed to run it. The name `web` is important here. It declares that this process type will be attached to the HTTP routing (https://devcenter.heroku.com/articles/http-routing) stack of Heroku, and receive web traffic when deployed.

Procfiles can contain additional process types. For example, you might declare one for a background worker process that processes items off of a queue.

Scale the app

Right now, your app is running on a single web dyno (https://devcenter.heroku.com/articles/dynos). Think of a dyno as a lightweight container that runs the command specified in the `Procfile`.

You can check how many dynos are running using the `ps` command:

```
$ heroku ps
=== web (Free): `node index.js`
web.1: up 2014/04/25 16:26:38 (~ 1s ago)
```

By default, your app is deployed on a free dyno. Free dynos will sleep after a half hour of inactivity (if they don't receive any traffic). This causes a delay of a few seconds for the first request upon waking. Subsequent requests will perform normally. Free dynos also consume from a monthly, account-level quota of free dyno hours (https://devcenter.heroku.com/articles/free-dyno-hours) - as long as the quota is not exhausted, all free apps can continue to run.

To avoid dyno sleeping, you can upgrade to a hobby or professional dyno type as described in the Dyno Types (https://devcenter.heroku.com/articles/dyno-types) article. For example, if you migrate your app to a professional dyno, you can easily scale it by running a command telling Heroku to execute a specific number of dynos, each running your web process type.

Scaling an application on Heroku is equivalent to changing the number of dynos that are running. Scale the number of web dynos to zero:

```
$ heroku ps:scale web=0
```

Access the app again by hitting refresh on the web tab, or `heroku open` to open it in a web tab. You will get an error message because you no longer have any web dynos available to serve requests.

Scale it up again:

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

For abuse prevention, scaling a non-free application to more than one dyno requires account verification (https://devcenter.heroku.com/articles/account-verification).

Declare app dependencies

Heroku recognizes an app as Node.js by the existence of a `package.json` file in the root directory. For your own apps, you can create one by running `npm init --yes`.

The demo app you deployed already has a `package.json`, and it looks something like this:

```
{
  "name": "node-js-getting-started",
  "version": "0.3.0",
  ...
  "engines": {
    "node": "12.x"
  },
  "dependencies": {
    "ejs": "^2.5.6",
    "express": "^4.15.2"
  },
  ...
}
```

The `package.json` file determines both the version of Node.js that will be used to run your application on Heroku, as well as the dependencies that should be installed with your application. When an app is deployed, Heroku reads this file and installs the appropriate node version together with the dependencies using the `npm install` command.

Run this command in your local directory to install the dependencies, preparing your system for running the app locally:

```
$ npm install
added 132 packages in 3.368s
```

Once dependencies are installed, you will be ready to run your app locally.

Run the app locally

Now start your application locally using the `heroku local` command, which was installed as part of the Heroku CLI:

```
$ heroku local web
[OKAY] Loaded ENV .env File as KEY=VALUE Format
1:23:15 PM web.1 |  Node app is running on port 5000
```

Just like Heroku, `heroku local` examines the `Procfile` to determine what to run.

Open http://localhost:5000 (http://localhost:5000) with your web browser. You should see your app running locally.

To stop the app from running locally, in the CLI, press `Ctrl` + `C` to exit.

Push local changes

In this step you'll learn how to propagate a local change to the application through to Heroku. As an example, you'll modify the application to add an additional dependency and the code to use it.

Begin by adding a dependency for `cool-ascii-faces` in `package.json`. Run the following command to do this:

```
$ npm install cool-ascii-faces
+ cool-ascii-faces@1.3.4
added 9 packages in 2.027s
```

Modify `index.js` so that it `requires` this module at the start. Also add a new route (`/cool`) that uses it. Your final code should look like this:

```
const cool = require('cool-ascii-faces')
const express = require('express')
const path = require('path')
const PORT = process.env.PORT || 5000

express()
  .use(express.static(path.join(__dirname, 'public')))
  .set('views', path.join(__dirname, 'views'))
  .set('view engine', 'ejs')
  .get('/', (req, res) => res.render('pages/index'))
  .get('/cool', (req, res) => res.send(cool()))
  .listen(PORT, () => console.log(`Listening on ${ PORT }`))
```

Now test locally:

```
$ npm install
$ heroku local
```

Visiting your application at http://localhost:5000/cool (http://localhost:5000/cool), you should see cute faces displayed on each refresh: `( ʘ _ ʘ )`.

Now deploy. Almost every deploy to Heroku follows this same pattern. First, add the modified files to the local git repository:

```
$ git add .
```

Now commit the changes to the repository:

```
$ git commit -m "Add cool face API"
```

Now deploy, just as you did previously:

```
$ git push heroku master
```

Finally, check that everything is working:

```
$ heroku open cool
```

You should see another face.

Provision add-ons

Add-ons are third-party cloud services that provide out-of-the-box additional services for your application, from persistence through logging to monitoring and more.

By default, Heroku stores 1500 lines of logs from your application. However, it makes the full log stream available as a service - and several add-on providers have written logging services that provide things such as log persistence, search, and email and SMS alerts.

In this step you will provision one of these logging add-ons, Papertrail.

Provision the papertrail (https://devcenter.heroku.com/articles/papertrail) logging add-on:

```
$ heroku addons:create papertrail
Adding papertrail on sharp-rain-871... done, v4 (free)
Welcome to Papertrail. Questions and ideas are welcome (support@papertrailapp.com). Happy logging!
Use `heroku addons:docs papertrail` to view documentation.
```

To help with abuse prevention, provisioning an add-on requires account verification (https://devcenter.heroku.com/articles/account-verification). If your account has not been verified, you will be directed to visit the verification site (https://heroku.com/verify).

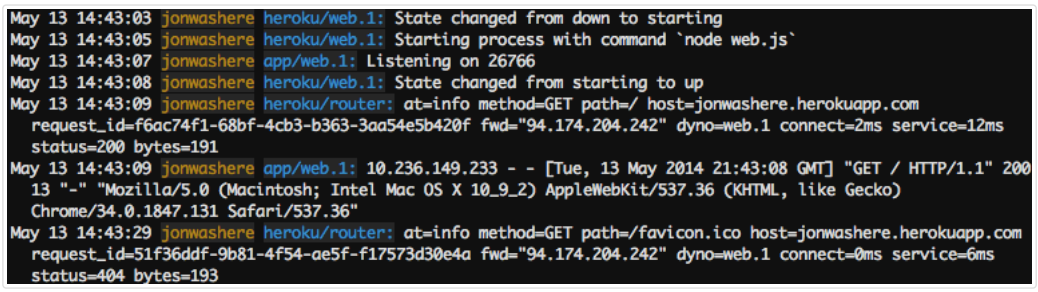
The add-on is now deployed and configured for your application. You can list add-ons for your app like so:

```
$ heroku addons
```

To see this particular add-on in action, visit your application's Heroku URL a few times. Each visit will generate more log messages, which should now get routed to the papertrail add-on. Visit the papertrail console to see the log messages:

```
$ heroku addons:open papertrail
```

Your browser will open up a Papertrail web console, showing the latest log events. The interface lets you search and set up alerts:



Start a console

To get a real feel for how dynos work, you can create another one-off dyno and run the `bash` command, which opens up a shell on that dyno. You can then execute commands there. Each dyno has its own ephemeral filespace, populated with your app and its dependencies - once the command completes (in this case, `bash`), the dyno is removed.

```
$ heroku run bash
Running `bash` attached to terminal... up, run.3052
~ $ ls
Procfile  README.md  composer.json  composer.lock  vendor  views  web
~ $ exit
exit
```

If you receive an error, `Error connecting to process`, then you may need to configure your firewall (https://devcenter.heroku.com/articles/one-off-dynos#timeout-awaiting-process).

Don't forget to type `exit` to exit the shell and terminate the dyno.

Define config vars

Heroku lets you externalize configuration - storing data such as encryption keys or external resource addresses in config vars (https://devcenter.heroku.com/articles/config-vars).

At runtime, config vars are exposed as environment variables to the application. For example, modify `index.js` so that it introduces a new route, `/times`, that repeats an action depending on the value of the `TIMES` environment variable. Under the existing `get()` call, add another:

```
.get('/times', (req, res) => res.send(showTimes()))
```

At the end of the file, add the following definition for the new function, `showTimes()`:

```
showTimes = () => {
  let result = ''
  const times = process.env.TIMES || 5
  for (i = 0; i < times; i++) {
    result += i + ' '
  }
  return result;
}
```

`heroku local` will automatically set up the environment based on the contents of the `.env` file in your local directory. In the top-level directory of your project, there is already a `.env` file that has the following contents:

```
TIMES=2
```

If you run the app with `heroku local`, you'll see two numbers will be generated every time.

To set the config var on Heroku, execute the following:

```
$ heroku config:set TIMES=2
```

View the config vars that are set using `heroku config`:

```
$ heroku config
== sharp-rain-871 Config Vars
PAPERTRAIL_API_TOKEN: erdKhPeeeehIcdfY7ne
TIMES: 2
```

Deploy your changed application to Heroku and then visit it by running `heroku open times`.

Provision a database

The add-on marketplace (https://elements.heroku.com/addons/categories/data-stores) has a large number of data stores, from Redis and MongoDB providers, to Postgres and MySQL. In this step, you will add a free Heroku Postgres Starter Tier dev database to your app.

Add the database:

```
$ heroku addons:create heroku-postgresql:hobby-dev
Adding heroku-postgresql:hobby-dev... done, v3 (free)
```

This creates a database, and sets a `DATABASE_URL` environment variable (you can check by running `heroku config`).

Use `npm` to install the pg module (https://npmjs.org/package/pg) to your dependencies:

```
$ npm install pg
+ pg@7.12.1
added 14 packages in 2.108s
```

```
  "dependencies": {
    "cool-ascii-faces": "^1.3.4",
    "ejs": "^2.5.6",
    "express": "^4.15.2",
    "pg": "^7.12.1"
  },
```

Now edit your `index.js` file to use this module to connect to the database specified in your DATABASE_URL environment variable. Add this near the top:

```
const { Pool } = require('pg');
const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: true
});
```

Now add another route, `/db`, by adding the following just after the existing `.get('/', ...)`:

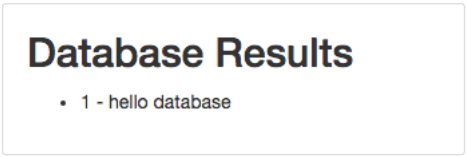
```
.get('/db', async (req, res) => {
    try {
      const client = await pool.connect()
      const result = await client.query('SELECT * FROM test_table');
      const results = { 'results': (result) ? result.rows : null};
      res.render('pages/db', results );
      client.release();
    } catch (err) {
      console.error(err);
      res.send("Error " + err);
    }
  })
```

This ensures that when you access your app using the `/db` route, it will return all rows in the `test_table` table.

Deploy this to Heroku. If you access `/db` you will receive an error as there is no table in the database. Assuming that you have Postgres installed locally (https://devcenter.heroku.com/articles/heroku-postgresql#local-setup), use the `heroku pg:psql` command to connect to the remote database, create a table and insert a row:

```
$ heroku pg:psql
psql (11.5)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.
=> create table test_table (id integer, name text);
CREATE TABLE
=> insert into test_table values (1, 'hello database');
INSERT 0 1
=> \q
```

Now when you access your app's `/db` route, you will see something like this:



Read more about Heroku PostgreSQL (https://devcenter.heroku.com/articles/heroku-postgresql).

A similar technique can be used to install MongoDB or Redis add-ons (https://elements.heroku.com/addons/categories/data-stores).

Next steps

You now know how to deploy an app, change its configuration, view logs, scale, and attach add-ons.

Here's some recommended reading. The first, an article, will give you a firmer understanding of the basics. The second is a pointer to the main Node.js category here on Dev Center:

- Read How Heroku Works (https://devcenter.heroku.com/articles/how-heroku-works) for a technical overview of the concepts you'll encounter while writing, configuring, deploying and running applications.
- Visit the Node.js category (https://devcenter.heroku.com/categories/nodejs-support) to learn more about developing and deploying Node.js applications.
- Read Deploying Node.js Apps on Heroku (https://devcenter.heroku.com/articles/deploying-nodejs) to understand how to take an existing Node.js app and deploy it to Heroku.