Getting Started on Heroku with Ruby

Last updated 30 September 2019

Table of Contents

Introduction

This tutorial will have you deploying a Ruby app in minutes.

Hang on for a few more minutes to learn how it all works, so you can make the most out of Heroku.

The tutorial assumes that you have:

- a free Heroku account (https://signup.heroku.com/dc).
- Ruby 2.3.5 installed locally - see the installation guides for Ruby and Rails on OS X (http://guides.railsgirls.com/install#setup-for-os-x), Windows (http://guides.railsgirls.com/install#setup-for-windows), and Linux (http://guides.railsgirls.com/install#setup-for-linux).
- Bundler (http://bundler.io/) installed locally - run `gem install bundler`.

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).

Once installed, you can use the `heroku` command from your command shell.

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.

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

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

You now have a functioning git repository that contains a simple application as well as a `Gemfile` file, which is used by Ruby's dependency manager, bundler.

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 app... done, ⬢ polar-inlet-4930
http://polar-inlet-4930.herokuapp.com/ | https://git.heroku.com/polar-inlet-4930.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 `polar-inlet-4930`) for your app, or you can pass a parameter to specify your own app name.

Now deploy your code:

```
$ git push heroku master
Enumerating objects: 365, done.
Counting objects: 100% (365/365), done.
Delta compression using up to 8 threads
Compressing objects: 100% (204/204), done.
Writing objects: 100% (365/365), 71.57 KiB | 35.79 MiB/s, done.
Total 365 (delta 141), reused 358 (delta 136)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Ruby app detected
remote: -----> Compiling Ruby/Rails
remote: -----> Using Ruby version: ruby-2.4.4
remote: -----> Installing dependencies using bundler 1.15.2
       ....
remote:        Asset precompilation completed (5.73s)
remote:        Cleaning assets
remote:        Running: rake assets:clean
remote: -----> Detecting rails configuration
remote:
remote:
remote: -----> Discovering process types
remote:        Procfile declares types     -> web
remote:        Default types for buildpack -> console, rake
remote:
remote: -----> Compressing...
remote:        Done: 74.3M
remote: -----> Launching...
remote:        Released v6
remote:        https://polar-inlet-4930.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/polar-inlet-4930.git
 * [new branch]      master -> master
```

The application is now deployed.

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
2019-09-18T14:10:10.769652+00:00 heroku[web.1]: Starting process with command `bundle exec puma -C config/puma.rb`
2019-09-18T14:10:13.134654+00:00 app[web.1]: [4] Puma starting in cluster mode...
2019-09-18T14:10:13.134680+00:00 app[web.1]: [4] * Version 4.1.1 (ruby 2.4.4-p296), codename: Fourth and One
2019-09-18T14:10:13.134682+00:00 app[web.1]: [4] * Min threads: 5, max threads: 5
2019-09-18T14:10:13.134685+00:00 app[web.1]: [4] * Environment: production
2019-09-18T14:10:13.134689+00:00 app[web.1]: [4] * Process workers: 2
2019-09-18T14:10:13.134722+00:00 app[web.1]: [4] * Preloading application

2019-09-18T14:10:15.517611+00:00 app[web.1]: [4] * Listening on tcp://0.0.0.0:48362
2019-09-18T14:10:15.785166+00:00 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: bundle exec puma –C config/puma.rb
```

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. The command used here is to run puma (the web server), passing in a configuration file.

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): `bundle exec puma -C config/puma.rb`
web.1: up 2015/05/12 11:28:21 (~ 4m 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 a Ruby app by the existence of a `Gemfile` file in the root directory.

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

```
source 'https://rubygems.org'
ruby '>= 2.3.5', '< 2.7'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '5.2.3'
# Use postgresql as the database for Active Record
gem 'pg', '~> 0.18'
# Use SCSS for stylesheets
gem 'sass-rails'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier'
...
```

The `Gemfile` file specifies the dependencies that should be installed with your application. You can also use it to determine the version of Ruby that will be used to run your application on Heroku.

When an app is deployed, Heroku reads this file and installs the appropriate Ruby version together with the dependencies using the `bundle install` command.

A prerequisite to running any app locally is to install the dependencies locally as well. This particular Gemfile has a dependency `pg` which will only resolve if you have Postgres installed locally (https://devcenter.heroku.com/articles/heroku-postgresql#local-setup). Please install Postgres before you proceed.

You will know that Postgres is installed successfully when the command `which psql` returns a value on your command line (though your actual output may vary):

```
$ which psql
/usr/local/bin/psql
```

Now run `bundle install` in your local directory to install the dependencies, preparing your system for running the app locally:

```
$ bundle install
Using rake 12.3.3
Using concurrent-ruby 1.1.5
Using i18n 1.6.0
....
Using turbolinks 5.2.0
Using uglifier 4.1.20
Your bundle is complete!
```

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

Run the app locally

Running apps locally in your own dev environment does require some effort. Rails typically requires a database - and this sample application uses Postgres. Check out these instructions (https://devcenter.heroku.com/articles/heroku-postgresql#local-setup) for a local install.

The Rails app also uses a database, so you'll have to create the appropriate database and table using the rake task:

```
$ bundle exec rake db:create db:migrate
== 20140707111715 CreateWidgets: migrating ================================
-- create_table(:widgets)
   -> 0.0076s
== 20140707111715 CreateWidgets: migrated (0.0077s) =======================
```

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

```
$ heroku local web
2:25:39 PM web.1 |  [69863] Puma starting in cluster mode...
2:25:39 PM web.1 |  [69863] * Version 4.1.1 (ruby 2.6.3-p62), codename: Fourth and One
2:25:39 PM web.1 |  [69863] * Min threads: 5, max threads: 5
2:25:39 PM web.1 |  [69863] * Environment: development
2:25:39 PM web.1 |  [69863] * Process workers: 2
2:25:39 PM web.1 |  [69863] * Preloading application
2:25:40 PM web.1 |  [69863] * Listening on tcp://0.0.0.0:5000
2:25:40 PM web.1 |  [69863] Use Ctrl-C to stop
2:25:40 PM web.1 |  [69863] - Worker 0 (pid: 69864) booted, phase: 0
2:25:40 PM web.1 |  [69863] - Worker 1 (pid: 69865) booted, phase: 0
```

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.

Modify `Gemfile` to include a dependency for the `cowsay` gem by including a line `gem 'cowsay'` :

```
source 'https://rubygems.org'
ruby '>= 2.3.5', '< 2.7'

gem 'cowsay'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '5.2.3'
...
```

Modify `app/views/welcome/index.erb` so that it uses this gem, by changing the file so that its first few lines read as follows:

```
<pre>
<%= Cowsay.say("Hello", "Cow") %>
</pre>
...
```

Now test locally:

```
$ bundle install
$ heroku local
```

Visit your application at http://localhost:5000 (http://localhost:5000). You should see a cute ASCII picture displayed.

Now deploy this local change to Heroku.

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 "Demo"
```

Now deploy, just as you did previously:

```
$ git push heroku master
```

Finally, check that everything is working:

```
$ heroku open
```

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 polar-inlet-4930... done, v11 (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).

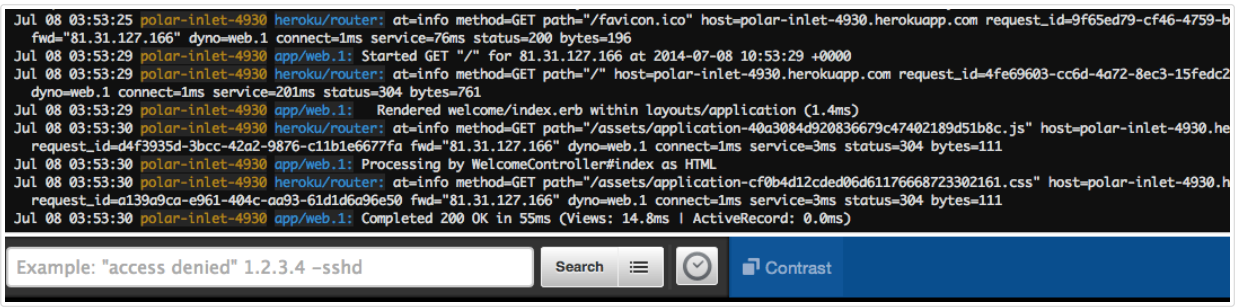
The add-on is now deployed and configured for your application. You can list add-ons for your app like so:

```
$ 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

You can run a command, typically scripts and applications that are part of your app, in a one-off dyno (https://devcenter.heroku.com/articles/one-off-dynos) using the `heroku run` command. It can also be used to launch a REPL process attached to your local terminal for experimenting in your app's environment:

```
$ heroku run rails console
Running rails console on ⬢ polar-inlet-4930... up, run.1360 (Free)
Loading production environment (Rails 5.2.3)
irb(main):001:0>
```

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).

When the console starts, it has your entire app loaded. For example, you can type `puts Cowsay.say("hi","Cow")` and an animal saying "hi" will be displayed. Type `exit` to quit the console.

```
irb(main):001:0> puts Cowsay.say("hi", "Cow")
 ____
| hi |
 ----
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/
=> nil
irb(main):002:0> exit
```

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.1421
~ $ ls
app  config db   Gemfile.lock  log   public    README.md  tmp
bin  config.ru  Gemfile  lib         Procfile  Rakefile  test   vendor
~ $ exit
exit
```

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 `app/views/welcome/index.erb` so that the method repeats an action depending on the value of the `TIMES` environment variable. Change the file so that its first few lines read as follows:

```
<% for i in 0..(ENV['TIMES'] ? ENV['TIMES'].to_i : 2) do %>
  <p>Hello World #<%= i %>!</p>
<% end %>
```

`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=10
```

If you run the app with `heroku local`, you'll see "Hello World" ten times.

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

```
$ heroku config:set TIMES=10
```

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

```
$ heroku config
== polar-inlet-4930 Config Vars
PAPERTRAIL_API_TOKEN: erdKhPeeeehIcdfY7ne
TIMES: 10
```

Deploy your changed application to Heroku to see this in action.

Use 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 learn about the free Heroku Postgres add-on that is provisioned automatically on all Rails app deploys.

A database is an add-on, and so you can find out a little more about the database provisioned for your app using the `addons` command in the CLI:

```
$ heroku addons
Add-on                                          Plan       Price  State
──────────────────────────────────────────────  ─────────  ─────  ───────
heroku-postgresql (postgresql-encircled-75487)  hobby-dev  free   created
 └─ as DATABASE
papertrail (papertrail-asymmetrical-81999)      choklad    free   created
 └─ as PAPERTRAIL

The table above shows add-ons and the attachments to the current app (polar-inlet-4930) or other apps.
```

Listing the config vars for your app will display the URL that your app is using to connect to the database, `DATABASE_URL`:

```
$ heroku config
=== polar-inlet-4930 Config Vars
DATABASE_URL:                postgres://xx:yyy@host:5432/d8slm9t7b5mjnd
HEROKU_POSTGRESQL_BROWN_URL: postgres://xx:yyy@host:5432/d8slm9t7b5mjnd
...
```

Heroku also provides a `pg` command that shows a lot more:

```
$ heroku pg
=== DATABASE_URL
Plan:                  Hobby-dev
Status:                Available
Connections:           0/20
PG Version:            11.5
Created:               2019-09-18 13:22 UTC
Data Size:             8.0 MB
Tables:                3
Rows:                  0/10000 (In compliance) - refreshing
Fork/Follow:           Unsupported
Rollback:              Unsupported
Continuous Protection: Off
Add-on:                postgresql-elliptical-30581
```

This indicates I have a hobby database (free), running Postgres 11.5, with a single row of data.

The example app you deployed already has database functionality - it has a controller and database model for widgets - which you should be able to reach by visiting your app's URL and appending '/widgets'.

If you do visit the URL, you'll see an error page appear. Check out the error message in using `heroku logs` or in Papertrail, and you'll see something like this:

```
2014-07-08T14:52:37.884178+00:00 app[web.1]: Started GET "/widgets" for 94.174.204.242 at 2014-07-08 14:52:37 +0000
2014-07-08T14:52:38.162312+00:00 heroku[router]: at=info method=GET path="/widgets" host=fox828228.herokuapp.com request_id=3755bb46-4de2-44
34-a13a-26ec73e53694 fwd="94.174.204.242" dyno=web.1 connect=0 service=294 status=500 bytes=955
2014-07-08T14:52:38.078295+00:00 app[web.1]: Processing by WidgetsController#index as HTML
....
2014-07-08T14:52:38.146062+00:00 app[web.1]: PG::UndefinedTable: ERROR:  relation "widgets" does not exist
```

This indicates that while we could connect to the database, the necessary table wasn't found. In Rails, you can fix that by running `rake db:migrate`. To execute this command on Heroku, run it in a one-off dyno like so:

```
$ heroku run rake db:migrate
Running `rake db:migrate` attached to terminal... up, run.3559
Migrating to CreateWidgets (20140707111715)
== 20140707111715 CreateWidgets: migrating ====================================
-- create_table(:widgets)
   -> 0.0244s
== 20140707111715 CreateWidgets: migrated (0.0247s) ===========================
```

Just like your web process type runs in a dyno, so too did this rake command. Heroku effectively boots a new dyno, adds in your prepared app, and then executes the command in that context - and afterwards removes the dyno. As the dyno performs its actions on a connected add-on, that's just perfect.

Now if you visit the `/widgets` page of your app again, you'll be able to list and create Widget records.

You can also interact directly with the database if you have Postgres installed locally. For example, here's how to connect to the database using `psql` and execute a query:

```
$ heroku pg:psql

d8slm9t7b5mjnd=> select * from widgets;
 id |   name    | description  | stock |         created_at         |         updated_at
----+-----------+--------------+-------+----------------------------+----------------------------
  1 | My Widget | It's amazing |   100 | 2014-07-08 15:05:13.330566 | 2014-07-08 15:05:13.330566
(1 row)
```

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

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:

- 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 Ruby support category (https://devcenter.heroku.com/categories/ruby-support) to learn more about developing and deploying Ruby applications.
- Read Getting Started with Rails 5.x on Heroku (https://devcenter.heroku.com/articles/getting-started-with-rails5) for a deep dive into deploying Rails apps.