We use cookies to make interactions with our websites and services easy and meaningful, to better understand how they are used and to tailor advertising. You can read more (https://www.salesforce.com/company/privacy/full_privacy.jsp#nav_info) and make your cookie choices here (https://www.salesforce.com/company/privacy/full_privacy.jsp#nav_info). By continuing to use this site you are giving us your consent to do this.

Getting Started on Heroku with Ruby (Microsoft Windows)

Last updated 03 October 2019

Table of Contents

Introduction

This tutorial will show you how to develop a Ruby app in a local Microsoft Windows development environment and then deploy it to Heroku. It uses JRuby, which provides a slightly more Windows-friendly environment.

If you're not on Windows, we recommend the traditional Getting Started with Ruby on Heroku (https://devcenter.heroku.com/articles/getting-started-with-ruby) guide.

The first step is to sign up for a free Heroku account (https://signup.heroku.com/signup/dc).

Next, install the prerequisites:

1. Download and install JDK 8 (https://www.oracle.com/technetwork/java/javase/downloads/index.html) to get a Java virtual machine (JVM).

2. Download and install JRuby 9.1.17.0 (https://www.jruby.org/download). Select the JRuby `.exe` installer for your version of Windows. After installing, start a new command prompt as your PATHs will have changed.
3. Install Bundler. To do this, open a command prompt and type: `jruby -S gem install bundler`.

If you receive an error such as `'jruby' is not recognized as an internal or external command, operable program or batch file.`, close your terminal session and open a new terminal to ensure that the `%PATH%` environment variable is set correctly.

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.

Windows

Download the appropriate installer for your Windows installation:

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

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

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.

Prepare the app

In this step, you will prepare a simple application that can be deployed.

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/jruby-getting-started.git
> cd jruby-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 polar-inlet-4930... done, stack is heroku-18
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.

Now, deploy your code:

```
> git push heroku master
Counting objects: 176, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (113/113), done.
Writing objects: 100% (176/176), 31.81 KiB | 0 bytes/s, done.
Total 176 (delta 51), reused 170 (delta 48)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Ruby app detected
remote: -----> Compiling Ruby/Rails
remote: -----> Using Ruby version: ruby-2.3.1-jruby-9.1.17.0
remote: -----> Installing JVM: openjdk1.8-latest
remote: Picked up JAVA_TOOL_OPTIONS: -Xmx768m -Djava.rmi.server.useCodebaseOnly=true
remote: -----> Installing dependencies using 1.7.12
remote:        Running: bundle install --without development:test --path vendor/bundle --bi
remote:        Picked up JAVA_TOOL_OPTIONS: -Xmx768m -Djava.rmi.server.useCodebaseOnly=true
remote:        Fetching gem metadata from https://rubygems.org/...........
remote:        Installing json 1.8.1
...
remote:        Installing sass-rails 4.0.5
remote:        Your bundle is complete!
remote:        Gems in the groups development and test were not installed.
remote:        It was installed into ./vendor/bundle
remote:        Bundle completed (162.29s)
remote:        Cleaning up the bundler cache.
remote: -----> Preparing app for Rails asset pipeline
remote:        Running: rake assets:precompile
...
remote:        Asset precompilation completed (107.72s)
remote:        Cleaning assets
remote:        Running: rake assets:clean
remote:        Picked up JAVA_TOOL_OPTIONS: -Xmx768m -Djava.rmi.server.useCodebaseOnly=true
remote:
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:        Default types for Ruby  -> console, rake, worker
remote:
remote: -----> Compressing... done, 97.9MB
remote: -----> Launching... done, 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. 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, `heroku logs --tail`:

```
> heroku logs --tail
2014-07-07T11:42:26.829065+00:00 heroku[web.1]: Starting process with command `bundle exec p
2014-07-07T11:42:35.334415+00:00 app[web.1]: I, [2014-07-07T11:42:35.334301 #2]  INFO -- : T
2014-07-07T11:42:35.707657+00:00 app[web.1]: I, [2014-07-07T11:42:35.707293 #5]  INFO -- : W
2014-07-07T11:42:35.772074+00:00 app[web.1]: I, [2014-07-07T11:42:35.771727 #11]  INFO -- :
2014-07-07T11:42:35.767750+00:00 app[web.1]: I, [2014-07-07T11:42:35.764688 #2]  INFO -- : n
2014-07-07T11:42:35.777268+00:00 app[web.1]: I, [2014-07-07T11:42:35.777006 #8]  INFO -- : W
2014-07-07T11:42:35.618291+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

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

To view the Procfile in the example app you deployed, type `type Procfile`. You will see that it looks like this: `web: puma -t 5:5 -p ${PORT:-3000} -e ${RACK_ENV:-development}`.

The Procfile 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 stack (https://devcenter.heroku.com/articles/http-routing) 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. 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): `web: puma -t 5:5 -p ${PORT:-3000} -e ${RACK_ENV:-development}`
web.1: up 2014/07/07 12:42:34 (~ 23m 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.

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 '2.3.1', :engine => 'jruby', :engine_version => '9.1.17.0'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.2.4'
# Use postgresql as the database for Active Record
gem 'activerecord-jdbcpostgresql-adapter'
gem 'rails_12factor', group: :production
gem 'puma'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.3'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
...
```

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.

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

```
> jruby -S bundle install --binstubs
Using rake 10.3.2
Using i18n 0.6.9
Using json 1.8.1
Using minitest 5.3.5
....
Using puma 2.11.0
Your bundle is complete!
```

If you receive the error `No such file or directory -- bundle (LoadError)`, then make sure you have run `jruby -S gem install bundler` as described in the introduction.

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. This sample application uses Postgres. You will need to follow the instructions on Dev Center for installing Postgres locally (https://devcenter.heroku.com/articles/heroku-postgresql#local-setup). When installing Postgres, make note of the password you set for the default user.

Open the `config\database.yml` file and set the username and password for your local (development) database. Look for these line:

```
development:
  <<: *default
  database: ruby-getting-started_development

  # The specified database role being used to connect to postgres.
  # To create additional roles in postgres see `$ createuser --help`.
  # When left blank, postgres will use the default role. This is
  # the same name as the operating system user that initialized the database.
  #username: ruby-getting-started

  # The password associated with the postgres role (username).
  #password:
```

Modify the username and password like this (but replace the password with the one you used upon installing Postgres):

```
  username: postgres

  # The password associated with the postgres role (username).
  password: postgres
```

Repeat this for the `ruby-getting-started_test` database, which will be in the `test:` section below the development database entry.

Now you can create the appropriate database and tables for the app using this rake task:

```
> jruby -S bin\rake db:create db:migrate
== 20140707111715 CreateWidgets: migrating ====================================
-- create_table(:widgets)
   -> 0.0076s
== 20140707111715 CreateWidgets: migrated (0.0077s) ===========================
```

The example project also contains a `Procfile.windows`, which contains the line, `web: jruby -S bin\puma -t 5:5 -p %PORT% -e development`.

This file is necessary because the command used to run the application on Windows is different from the command used to run the application on Heroku, which is Linux-based. We'll use this file later in the tutorial.

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

```
> heroku local web -f Procfile.windows
13:15:47 web.1  | started with pid 67489
13:15:47 web.1  | I, [2014-07-07T13:15:47.655153 #67489]  INFO -- : Refreshing Gem list
13:15:48 web.1  | I, [2014-07-07T13:15:48.495226 #67489]  INFO -- : listening on addr=0.0.0
13:15:48 web.1  | I, [2014-07-07T13:15:48.621967 #67489]  INFO -- : master process ready
13:15:48 web.1  | I, [2014-07-07T13:15:48.624523 #67491]  INFO -- : worker=0 ready
13:15:48 web.1  | I, [2014-07-07T13:15:48.626285 #67492]  INFO -- : worker=1 ready
13:15:48 web.1  | I, [2014-07-07T13:15:48.627737 #67493]  INFO -- : worker=2 ready
```

The `-f Procfile.windows` flag ensures your Windows-specific Procfile is picked up. Just like Heroku, `heroku local` examines it 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, go back to your terminal window and 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. Your file will look something like this:

```
gem 'cowsay'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.2.4'
...
```

Modify `app/views/welcome/index.erb` so that it uses this gem. Your final code should look like this:

```
<h1>Getting Started with Ruby</h1>

<p>
  Welcome!
</p>

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

Now test locally:

```
> jruby -S bundle install
> heroku local web -f Procfile.windows
```

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

Now, deploy this local change to Heroku.

Almost every deploy to Heroku follows the same pattern. First, add the modified files to your 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 logging add-on:

```
> heroku addons:create papertrail
Creating papertrail on ⬢ tranquil-tor-77660... free
Welcome to Papertrail. Questions and ideas are welcome (support@papertrailapp.com). Happy lo
Created papertrail-rugged-93041 as PAPERTRAIL_API_TOKEN
Use heroku addons:docs papertrail to view documentation
```

To help with abuse prevention, provisioning an add-on requires account verification. If your account has not been verified, you will be directed to visit the verification site.

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 will 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 one-off dyno

You can run a command, typically scripts and applications that are part of your app, in a one-off dyno 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 jirb
Running `rails console` attached to terminal... up, run.1594
Loading production environment (Rails 4.2.4)
irb(main):001:0>
```

If you receive the error, "Error connecting to process", then you may need to configure your firewall.

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

```
irb(main):001:0> require 'cowsay'
require 'cowsay'
=> true
irb(main):002:0> puts Cowsay.say("Hello", "Cow")
 _______
| Hello |
 -------
      \   ^__^
       \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
=> nil
irb(main):003:0> 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. When 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.rdoc  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.

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.

```
<h1>Getting Started with Ruby</h1>

<p>
  Welcome!
</p>

<% 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, type 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 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 more about the database provisioned for your app using the `heroku addons` command in the CLI:

```
> heroku addons

Add-on                                          Plan       Price  State
──────────────────────────────────────────────  ─────────  ─────  ───────
heroku-postgresql (postgresql-concave-37514)  hobby-dev  free   created
 └─ as DATABASE

papertrail (papertrail-rugged-93041)          choklad    free   created
 └─ as PAPERTRAIL
```

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 `heroku pg` command that shows a lot more:

```
> heroku pg
=== HEROKU_POSTGRESQL_BROWN_URL (DATABASE_URL)
Plan:        Hobby-dev
Status:      Available
Connections: 0
PG Version:  9.3.3
Created:     2014-07-07 11:30 UTC
Data Size:   6.6 MB
Tables:      2
Rows:        1/10000 (In compliance)
Fork/Follow: Unsupported
Rollback:    Unsupported
```

This indicates I have a hobby database (free), running Postgres 9.3.3, 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 visit the URL, you'll see an error page appear. View the error message by using `heroku logs`. In Papertrail, 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=3755bb46-4de2-4434-a13a-26ec73e53694 fwd="94.174.204.2
42" 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 HTM
L
....
2014-07-08T14:52:38.146062+00:00 app[web.1]: PG::UndefinedTable: ERROR:  relation "widget
s" 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 this:

```
> 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 boots a new dyno, adds in your prepared app, and then executes the command in that context - and afterwards removes the dyno.

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 15:05:13.3
(1 row)
```

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/#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:

- 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 category (https://devcenter.heroku.com/categories/ruby-support) to learn more about developing and deploying Ruby applications.