# Getting Started on Heroku with PHP

Last updated 01 October 2019

**Table of Contents**



## Introduction

This tutorial will have you deploying a PHP 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/signup/dc).
- PHP (http://php.net/) installed locally.
- Composer (https://getcomposer.org/doc/00-intro.md) 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).

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.

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 if you have PHP installed - check that it's there:

```
$ php -v
PHP 7.0.5 (cli) (built: Apr 26 2016 04:39:48) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
```

Now check that you have `composer` installed. If not, install it (https://getcomposer.org/doc/00-intro.md) and test again:

```
$ composer -V
Composer version 1.4.1 2017-03-10 09:29:45
```

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

```
$ git --version
git version 2.12.2
```

## 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/php-getting-started.git
$ cd php-getting-started
```

You now have a functioning git repository that contains a simple application as well as a `composer.json` file. Make sure you've installed Composer (http://getcomposer.org). Heroku uses Composer for dependency management in PHP projects, and the `composer.json` file indicates to Heroku that your application is written in PHP.

## 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
remote: Building source:
remote:
remote: -----> PHP app detected
remote: -----> Bootstrapping...
remote: -----> Installing platform packages...
remote:        NOTICE: No runtime required in composer.json; requirements
remote:        from dependencies in composer.lock will be used for selection
remote:        - php (7.1.3)
remote:        - apache (2.4.20)
remote:        - nginx (1.8.1)
remote: -----> Installing dependencies...
remote:        Composer version 1.4.1 2017-03-10 09:29:45
remote:        Loading composer repositories with package information
remote:        Installing dependencies from lock file
remote:        Package operations: 12 installs, 0 updates, 0 removals
remote:          - Installing psr/log (1.0.2): Loading from cache
remote:          - Installing monolog/monolog (1.22.1): Loading from cache
...
remote:          - Installing symfony/twig-bridge (v3.2.7): Loading from cache
remote:        Generating optimized autoload files
remote: -----> Preparing runtime environment...
remote: -----> Checking for additional extensions to install...
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 14.8M
remote: -----> Launching...
remote:        Released v17
remote:        https://gsphpjon.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/gsphpjon.git
 + 264e577...4f2369c master -> master (forced update)
```

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
014-05-27T11:03:21.033331+00:00 app[web.1]: Booting on port 28661...
2014-05-27T11:03:21.127050+00:00 app[web.1]: Using Apache2 configuration file 'vendor/heroku/heroku-buildpack-php/conf/apache2/heroku.conf'
2014-05-27T11:03:21.068192+00:00 app[web.1]: Using PHP configuration (php.ini) file 'vendor/heroku/heroku-buildpack-php/conf/php/php.ini'
...
2014-05-27T11:03:23.883157+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.

```
2014-05-30T13:18:13.581545+00:00 app[web.1]: [30-May-2014 13:18:13] WARNING: [pool www] child 59 said into stderr: "[2014-05-30 13:18:13] myapp
```

Logging, then, is simply a matter of directing output to `stdout` or `stderr` - Heroku does the work of aggregating this across all the app and system components. View the `web/index.php` (https://github.com/heroku/php-getting-started/blob/master/web/index.php#L9) file to see how the `Monolog` service is configured to write its output to `stderr`.

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: vendor/bin/heroku-php-apache2 web/
```

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): `vendor/bin/heroku-php-apache2`
web.1: up 2014/05/27 12:03:23 (~ 11m 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 PHP by the existence of a `composer.json` file in the root directory.

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

```
{
  "require" : {
    "silex/silex": "^2.0.4",
    "monolog/monolog": "^1.22",
    "twig/twig": "^2.0",
    "symfony/twig-bridge": "^3"
  },
  "require-dev": {
    "heroku/heroku-buildpack-php": "*"
  }
}
```

The `composer.json` file specifies the dependencies that should be installed with your application. When an app is deployed, Heroku reads this file and installs the appropriate dependencies into the `vendor` directory.

Your PHP app can then make use of the dependencies after a simple require:

```
require('../vendor/autoload.php');
```

Run the following command to install the dependencies, preparing your system for running the app locally:

```
$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing psr/log (1.0.0)
    Loading from cache
...
Writing lock file
Generating autoload files
```

You should always check `composer.json` and `composer.lock` into your git repo. The `vendor` directory should be included in your `.gitignore` file.

## 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 (the Cowsay (https://github.com/alrik11es/cowsayphp) library) and the code to use it.

First, use composer to require the new dependency:

```
$ composer require alrik11es/cowsayphp
```

This will also change `composer.json`. If you introduced the dependency by modifying the `composer.json` file yourself, be sure to update the dependencies by running:

```
$ composer update
```

Now modify `web/index.php` to use this library. Add a new route after the existing one, for `/cowsay`:

```
$app->get('/cowsay', function() use($app) {
  $app['monolog']->addDebug('cowsay');
  return "<pre>".\Cowsayphp\Cow::say("Cool beans")."</pre>";
});
```

When that route is visited, it will render a beautiful cow.

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 "Demo"
```

Now deploy, just as you did previously:

```
$ git push heroku master
```

Finally, check that everything is working:

```
$ heroku open cowsay
```

## 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)
...
Happy logging!
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 (https://heroku.com/verify).

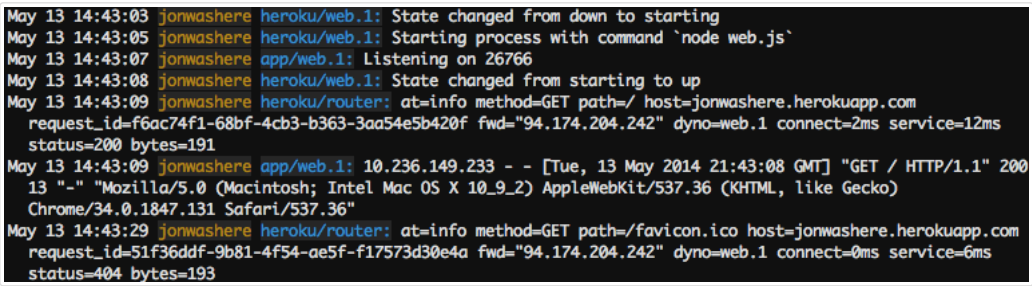
The add-on is now deployed and configured for your application. You can list add-ons for your app like so:

```
$ 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 an interactive shell

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 an interactive PHP shell attached to your local terminal for experimenting in your app's environment:

```
$ heroku run "php -a"
Running `php -a` attached to terminal... up, run.8081
Interactive shell

php > echo PHP_VERSION;
5.5.12
```

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).

The PHP console has nothing loaded other than the PHP standard library. To quit the PHP shell, type `quit`.

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
```

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

## Define config vars

Heroku lets you externalise 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.

Modify `web/index.php` so that root route returns the word `Hello` repeated by the value of the `TIMES` environment variable:

```
$app->get('/', function() use($app) {
  $app['monolog']->addDebug('logging output.');
  return str_repeat('Hello', getenv('TIMES'));
});
```

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

```
$ heroku config:set TIMES=20
```

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

```
$ heroku config
=== sharp-rain-871 Config Vars
PAPERTRAIL_API_TOKEN: erdKhPeeehIcdfY7ne
TIMES: 20
```

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

## 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` config var (you can check by running `heroku config`).

Modify `composer.json` to include a dependency for a simple PDO service provider, csanquer/pdo-service-provider (https://packagist.org/packages/csanquer/pdo-service-provider):

```
$ composer require csanquer/pdo-service-provider=~1.1dev
```

Install the new dependency:

```
$ composer update
```

Now modify `index.php` to extend the app to add a PDO connection:

```
$dbopts = parse_url(getenv('DATABASE_URL'));
$app->register(new Csanquer\Silex\PdoServiceProvider\Provider\PDOServiceProvider('pdo'),
               array(
                'pdo.server' => array(
                   'driver'   => 'pgsql',
                   'user' => $dbopts["user"],
                   'password' => $dbopts["pass"],
                   'host' => $dbopts["host"],
                   'port' => $dbopts["port"],
                   'dbname' => ltrim($dbopts["path"],'/')
                   )
               )
);
```

Note how this code retrieves the `DATABASE_URL` config var from the environment using `getenv()` (http://php.net/manual/function.getenv.php), and extracts information on hostname, database and credentials from that config var using `parse_url()` (http://php.net/manual/function.parse-url.php).

In the same file, add a new handler to query the database:

```
$app->get('/db/', function() use($app) {
  $st = $app['pdo']->prepare('SELECT name FROM test_table');
  $st->execute();

  $names = array();
  while ($row = $st->fetch(PDO::FETCH_ASSOC)) {
    $app['monolog']->addDebug('Row ' . $row['name']);
    $names[] = $row;
  }

  return $app['twig']->render('database.twig', array(
    'names' => $names
  ));
});
```

This ensures that when you access your app using the `/db` route, it will return all rows in the `test_table` table, and render the results using the `database.twig` template. Create the template in the `web/views` directory:

```
{% extends "layout.html" %}

{% block content %}
<p>Got these rows from the database:</p>

<ul>
{% for n in names %}
  <li> {{ n.name }} </li>
{% else %}
  <li>Nameless!</li>
{% endfor %}
</ul>

{% endblock %}
```

If you get lost making these changes, take a look at the db branch (https://github.com/heroku/php-getting-started/tree/db) of the sample app.

Deploy the app modifications to Heroku:

```
$ git add .
$ git commit -m "added database access"
$ git push heroku master
```

If you now access `/db` you will see `Nameless` in the output 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 database you provisioned earlier, create a table and insert a row:

```
$ heroku pg:psql
psql (9.5.2, server 9.6.2)
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:

```
Got these rows from the database:

* hello database
```

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:

- 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.
- Read Deploying PHP Apps on Heroku (https://devcenter.heroku.com/articles/deploying-php) to understand how to take an existing PHP app and deploy it to Heroku.
- Visit the PHP category (https://devcenter.heroku.com/categories/php-support) to learn more about developing and deploying PHP applications.