How Heroku Works

Last updated 26 August 2016

Table of Contents

This is a high-level, technical description of how Heroku works. It ties together many of the concepts you'll encounter while writing, configuring, deploying and running applications on the Heroku platform.

> Performing one of the **Getting Started (https://devcenter.heroku.com/start)** tutorials will make the concepts in this documentation more concrete.

Read this document sequentially: in order to tell a coherent story, it incrementally unveils and refines the concepts describing the platform.

The final section ties all the definitions together, providing a deploy-time and runtime-view of Heroku.

Defining an application

Heroku lets you deploy, run and manage applications written in Ruby, Node.js, Java, Python, Clojure, Scala, Go and PHP.

An application is a collection of *source code* written in one of these languages, perhaps a framework, and some *dependency description* that instructs a build system as to which additional dependencies are needed in order to build and run the application.

> **Terminology** (Preliminary): Applications consist of your source code and a description of any dependencies.

Dependency mechanisms vary across languages: in Ruby you use a `Gemfile`, in Python a `requirements.txt`, in Node.js a `package.json`, in Java a `pom.xml` and so on.

The source code for your application, together with the dependency file, should provide enough information for the Heroku platform to build your application, to produce something that can be executed.

Knowing what to execute

You don't need to make many changes to an application in order to run it on Heroku. One requirement is informing the platform as to which parts of your application are runnable.

If you're using some established framework, Heroku can figure it out. For example, in Ruby on Rails, it's typically `rails server`, in Django it's `python <app>/manage.py runserver` and in Node.js it's the `main` field in `package.json`.

Terminology: **Procfiles (https://devcenter.heroku.com/articles/procfile)** list process types - named commands that you may want executed.

For other applications, you may need to explicitly declare what can be executed. You do this in a text file that accompanies your source code - a Procfile (https://devcenter.heroku.com/articles/procfile). Each line declares a process type (https://devcenter.heroku.com/articles/process-model) - a named command that can be executed against your built application. For example, your Procfile may look like this:

```
web: java -jar lib/foobar.jar $PORT
queue: java -jar lib/queue-processor.jar
```

This file declares a `web` process type and provides the command that needs to be executed in order to run it (in this case, `java -jar lib/foobar.jar $PORT`). It also declares a `queue` process type, and its corresponding command.

The earlier definition of an application can now be refined to include this single additional Procfile.

Terminology: Applications consist of your source code, a description of any dependencies, and a Procfile.

Heroku is a polyglot platform – it lets you build, run and scale applications in a similar manner across all the languages – utilizing the dependencies and Procfile. The Procfile exposes an architectural aspect of your application (in the above example there are two entry points to the application) and this architecture lets you, for example, scale each part independently. An excellent guide to architecture principles that work well for applications running on Heroku can be found in Architecting Applications for Heroku (https://devcenter.heroku.com/articles/architecting-apps).

Deploying applications

Git (http://git-scm.com/) is a powerful, distributed version control system that many developers use to manage and version source code. The Heroku platform uses git as the primary means for deploying applications (there are other ways to transport your source code to Heroku, including via an API).

When you create an application on Heroku, it associates a new git remote, typically named `heroku`, with the local git repository for your application.

As a result, deploying code is just the familiar `git push`, but to the `heroku` remote instead:

```
$ git push heroku master
```

Terminology: Deploying applications involves sending the application to Heroku using either git, GitHub, Dropbox, or via an API.

There are many other ways of deploying applications too. For example, you can enable GitHub integration (https://devcenter.heroku.com/articles/github-integration) so that each new pull request is associated with its own new application, which enables all sorts of continuous integration scenarios. Or you can use Dropbox Sync (https://devcenter.heroku.com/articles/dropbox-sync), which lets you deploy the contents of Dropbox folders to Heroku. Finally, you can also use the Heroku API (https://devcenter.heroku.com/articles/build-and-release-using-the-api) to build and release apps.

Deployment then, is about moving your application from your local system to Heroku - and Heroku provides several ways in which apps can be deployed.

Building applications

When the Heroku platform receives the application source, it initiates a build of the source application. The build mechanism is typically language specific, but follows the same pattern, typically retrieving the specified dependencies, and creating any necessary assets (whether as simple as processing style sheets or as complex as compiling code).

Advanced: **Buildpacks (https://devcenter.heroku.com/articles/buildpacks)** lie behind the slug compilation process. Buildpacks take your application, its dependencies, and the language runtime, and produce slugs. They're open source - enabling you to extend Heroku to other languages and frameworks.

For example, when the build system receives a Rails application, it may fetch all the dependencies specified in the Gemfile, as well as generate files based on the asset pipeline. A Java application may fetch binary library dependencies using Maven, compile the source code together with those libraries, and produce a JAR file to execute.

The source code for your application, together with the fetched dependencies and output of the build phase such as generated assets or compiled code, as well as the language and framework, are assembled into a slug (https://devcenter.heroku.com/articles/slug-compiler).

Terminology: A **slug (https://devcenter.heroku.com/articles/slug-compiler)** is a bundle of your source, fetched dependencies, the language runtime, and compiled/generated output of the build system - ready for execution.

These slugs are a fundamental aspect of what happens during application execution - they contain your compiled, assembled application - ready to run - together with the instructions (the Procfile) of what you may want to execute.

Running applications on dynos

Heroku executes applications by running a command you specified in the Procfile, on a dyno (https://devcenter.heroku.com/articles/dynos) that's been preloaded with your prepared slug (in fact, with your release, which extends your slug and a few items not yet defined: config vars and add-ons).

Think of a running dyno as a lightweight, secure, virtualized Unix container that contains your application slug in its file system.

Terminology: **Dynos (https://devcenter.heroku.com/articles/dynos)** are isolated, virtualized Unix containers, that provide the environment required to run an application.

Generally, if you deploy an application for the first time, Heroku will run 1 web dyno automatically. In other words, it will boot a dyno, load it with your slug, and execute the command you've associated with the web process type in your Procfile.

You have control over how many dynos are running at any given time. Given the Procfile example earlier, you can start 5 dynos, 3 for the web and 2 for the queue process types, as follows:

```
$ heroku ps:scale web=3 queue=2
```

When you deploy a new version of an application, all of the currently executing dynos are killed, and new ones (with the new release) are started to replace them - preserving the existing dyno formation.

Terminology: Your application's **dyno formation (https://devcenter.heroku.com/articles/scaling#dyno-formation)** is the total number of currently-executing dynos, divided between the various process types you have scaled.

To understand what's executing, you just need to know what dynos are running which process types:

```
$ heroku ps
== web: 'java lib/foobar.jar $PORT'
web.1: up 2013/02/07 18:59:17 (~ 13m ago)
web.1: up 2013/02/07 18:52:08 (~ 20m ago)
web.2: up 2013/02/07 18:31:14 (~ 41m ago)

== queue: `java lib/queue-processor.jar`
queue.1: up 2013/02/07 18:40:48 (~ 32m ago)
queue.2: up 2013/02/07 18:40:48 (~ 32m ago)
```

Dynos then, are an important means of scaling your application. In this example, the application is well architected to allow for the independent scaling of web and queue worker dynos.

Config vars

An application's configuration is everything that is likely to vary between environments (staging, production, developer environments, etc.). This includes backing services such as databases, credentials, or environment variables that provide some specific information to your application.

Heroku lets you run your application with a customizable configuration - the configuration sits outside of your application code and can be changed independently of it.

The configuration for an application is stored in config vars (https://devcenter.heroku.com/articles/config-vars). For example, here's how to configure an encryption key for an application:

```
$ heroku config:set ENCRYPTION_KEY=my_secret_launch_codes
Adding config vars and restarting demoapp... done, v14
ENCRYPTION_KEY:     my_secret_launch_codes
```

Terminology: **Config vars (https://devcenter.heroku.com/articles/config-vars)** contain customizable configuration data that can be changed independently of your source code. The configuration is exposed to a running application via environment variables.

At runtime, all of the config vars are exposed as environment variables - so they can be easily extracted programatically. A Ruby application deployed with the above config var can access it by calling `ENV["ENCRYPTION_KEY"]`.

All dynos in an application will have access to the exact same set of config vars at runtime.

Releases

Earlier, this article stated that to run your application on a dyno, the Heroku platform loaded the dyno with your most recent slug. This needs to be refined: in fact it loads it with the slug and any config variables you have assigned to the application. The combination of slug and configuration is called a release (https://devcenter.heroku.com/articles/releases).

Terminology (Preliminary): **Releases (https://devcenter.heroku.com/articles/releases)** are an append-only ledger of slugs and config vars.

All releases are automatically persisted in an append-only ledger, making managing your application, and different releases, a cinch. Use the `heroku releases` command to see the audit trail of release deploys:

```
$ heroku releases
== demoapp Releases
v103 Deploy 582fc95  jon@heroku.com   2013/01/31 12:15:35
v102 Deploy 990d916  jon@heroku.com   2013/01/31 12:01:12
```

The number next to the deploy message, for example `582fc95`, corresponds to the commit hash of the repository you deployed to Heroku.

Every time you deploy a new version of an application, a new slug is created and release is generated.

As Heroku contains a store of the previous releases of your application, it's very easy to rollback and deploy a previous release:

```
$ heroku releases:rollback v102
Rolling back demoapp... done, v102
$ heroku releases
== demoapp Releases
v104 Rollback to v102 jon@heroku.com   2013/01/31 14:11:33 (~15s ago)
v103 Deploy 582fc95   jon@heroku.com   2013/01/31 12:15:35
v102 Deploy 990d916   jon@heroku.com   2013/01/31 12:01:12
```

Making a material change to your application, whether it's changing the source or configuration, results in a new release being created.

A release then, is the mechanism behind how Heroku lets you modify the configuration of your application (the config vars) independently of the application source (stored in the slug) - the release binds them together. Whenever you change a set of config vars associated with your application, a new release will be generated.

Dyno manager

Part of the Heroku platform, the dyno manager (https://devcenter.heroku.com/articles/dynos), is responsible for keeping dynos running. For example, dynos are cycled at least once per day, or whenever the dyno manager detects a fault in the running application (such as out of memory exceptions) or problems with the underlying hardware that requires the dyno be moved to a new physical location.

Terminology: The **dyno manager (https://devcenter.heroku.com/articles/dynos)** of the Heroku platform is responsible for managing dynos across all applications running on Heroku.

This dyno cycling happens transparently and automatically on a regular basis, and is logged.

Terminology: Applications that use the free dyno type will **sleep (https://devcenter.heroku.com/articles/free-dyno-hours)**. When a sleeping application receives HTTP traffic, it will be awakened - causing a delay of a few seconds. Using one of the other **dyno types (https://devcenter.heroku.com/articles/dyno-types)** will avoid sleeping.

Because Heroku manages and runs applications, there's no need to manage operating systems or other internal system configuration. One-off dynos (https://devcenter.heroku.com/articles/one-off-dynos) can be run with their input/output attached to your local terminal. These can also be used to carry out admin tasks that modify the state of shared resources, for example database configuration - perhaps periodically through a scheduler (https://devcenter.heroku.com/articles/scheduler).

Terminology: **One-off Dynos (https://devcenter.heroku.com/articles/one-off-dynos)** are temporary dynos that can run with their input/output attached to your local terminal. They're loaded with your latest release.

Here's the simplest way to create and attach to a one-off dyno:

```
$ heroku run bash
Running `bash` attached to terminal... up, run.8963
~ $ ls
```

This will spin up a new dyno, loaded with your release, and then run the `bash` command - which will provide you with a unix shell (remember that dynos are effectively isolated virtualized unix containers). Once you've terminated your session, or after a period of inactivity, the dyno will be removed.

Changes to the filesystem on one dyno are not propagated to other dynos and are not persisted across deploys and dyno restarts. A better and more scalable approach is to use a shared resource such as a database or queue.

Terminology: Each dyno gets its own **ephemeral filesystem (https://devcenter.heroku.com/articles/dynos#ephemeral-filesystem)** - with a fresh copy of the most recent release. It can be used as temporary scratchpad, but changes to the filesystem are not reflected to other dynos.

The ephemeral nature of the file system in a dyno can be demonstrated with the above command. If you create a one-off dyno by running `heroku run bash`, the Unix shell on the dyno, and then create a file on that dyno, and then terminate your session - the change is lost. All dynos, even those in the same application, are isolated - and after the session is terminated the dyno will be killed. New dynos are always created from a slug, not from the state of other dynos.

Add-ons

Applications typically make use of add-ons (https://devcenter.heroku.com/articles/add-ons) to provide backing services such as databases, queueing & caching systems, storage, email services and more. Add-ons are provided as services by Heroku and third parties - there's a large marketplace (https://elements.heroku.com/addons) of add-ons you can choose from.

Heroku treats these add-ons as attached resources: provisioning an add-on is a matter of choosing one from the add-on marketplace, and attaching it to your application.

For example, here is how to add the Heroku Redis (https://devcenter.heroku.com/articles/heroku-redis) backing store add-on to an application:

```
$ heroku addons:create heroku-redis:hobby-dev
```

Dynos do not share file state, and so add-ons that provide some kind of storage are typically used as a means of communication between dynos in an application. For example, Redis or Postgres could be used as the backing mechanism in a queue; then dynos of the web process type can push job requests onto the queue, and dynos of the queue process type can pull jobs requests from the queue.

The add-on service provider is responsible for the service - and the interface to your application is often provided through a config var. In this example, a `REDIS_URL` will be automatically added to your application when you provision the add-on. You can write code that connects to the service through the URL, for example:

```
uri = URI.parse(ENV["REDIS_URL"])
REDIS = Redis.new(:host => uri.host, :port => uri.port, :password => uri.password)
```

Terminology: **Add-ons (https://elements.heroku.com/addons/)** are third party, specialized, value-added cloud services that can be easily attached to an application, extending its functionality.

Add-ons are associated with an application, much like config vars - and so the earlier definition of a release needs to be refined. A *release* of your applications is not just your slug and config vars; it's your slug, config vars as well as the set of provisioned add-ons.

Terminology: **Releases (https://devcenter.heroku.com/articles/releases)** are an append-only ledger of slugs, config vars and add-ons. Heroku maintains an append-only ledger of releases you make.

Much like config vars, whenever you add, remove or change an add-on, a new release is created.

Logging and monitoring

Heroku treats logs as streams of time-stamped events, and collates the stream of logs produced from all of the processes running in all dynos, and the Heroku platform components, into the Logplex (https://devcenter.heroku.com/articles/logplex) - a high-performance, real-time system for log delivery.

It's easy to examine the logs across all the platform components and dynos:

```
$ heroku logs
2013-02-11T15:19:10+00:00 heroku[router]: at=info method=GET path=/articles/custom-domains host=mydemoa
2013-02-11T15:19:10+00:00 app[web.2]: Started GET "/" for 1.169.38.175 at 2013-02-11 15:19:10 +0000
2013-02-11T15:19:10+00:00 app[web.1]: Started GET "/" for 2.161.132.15 at 2013-02-11 15:20:10 +0000
```

Here you see 3 timestamped log entries, the first from Heroku's router, the last two from two dynos running the web process type.

Terminology: **Logplex (https://devcenter.heroku.com/articles/logplex)** automatically collates log entries from all the running dynos of your app, as well as other components such as the routers, providing a single source of activity.

You can also dive into the logs from just a single dyno, and keep the channel open, listening for further events:

```
$ heroku logs --ps web.1 --tail
2013-02-11T15:19:10+00:00 app[web.2]: Started GET "/" for 1.169.38.175 at 2013-02-11 15:19:10 +0000
```

Logplex keeps a limited buffer of log entries solely for performance reasons. To persist them, and action events such as email notification on exception, use a Logging Add-on (https://elements.heroku.com/addons/#logging), which ties into log drains - an API for receiving the output from Logplex.

HTTP routing

Depending on your dyno formation, some of your dynos will be running the command associated with the `web` process type, and some will be running other commands associated with other process types.

The dynos that run process types named `web` are different in one way from all other dynos - they will receive HTTP traffic. Heroku's HTTP routers (https://devcenter.heroku.com/articles/http-routing) distribute incoming requests for your application across your running web dynos.

So scaling an app's capacity to handle web traffic involves scaling the number of web dynos:

```
$ heroku ps:scale web+5
```

A random selection algorithm is used for HTTP request load balancing across web dynos - and this routing handles both HTTP and HTTPS traffic. It also supports multiple simultaneous connections, as well as timeout handling.

Tying it all together

The concepts explained here can be divided into two buckets: those that involve the development and deployment of an application, and those that involve the runtime operation of the Heroku platform and the application after it's deployed.

The following two sections recapitulate the main components of the platform, separating them into these two buckets.

Deploy

- Applications consist of your source code, a description of any dependencies, and a Procfile.
- Procfiles (https://devcenter.heroku.com/articles/procfile) list process types - named commands that you may want executed.
- Deploying applications involves sending the application to Heroku using either git, GitHub, Dropbox, or via an API.
- Buildpacks (https://devcenter.heroku.com/articles/buildpacks) lie behind the slug compilation process. Buildpacks take your application, its dependencies, and the language runtime, and produce slugs.
- A slug (https://devcenter.heroku.com/articles/slug-compiler) is a bundle of your source, fetched dependencies, the language runtime, and compiled/generated output of the build system - ready for execution.
- Config vars (https://devcenter.heroku.com/articles/config-vars) contain customizable configuration data that can be changed independently of your source code. The configuration is exposed to a running application via environment variables.
- Add-ons (https://elements.heroku.com/addons/) are third party, specialized, value-added cloud services that can be easily attached to an application, extending its functionality.

- A release (https://devcenter.heroku.com/articles/releases) is a combination of a slug (your application), config vars and add-ons. Heroku maintains an append-only ledger of releases you make.

Runtime

- Dynos (https://devcenter.heroku.com/articles/dynos) are isolated, virtualized unix containers, that provide the environment required to run an application.
- Your application's dyno formation (https://devcenter.heroku.com/articles/scaling#dyno-formation) is the total number of currently-executing dynos, divided between the various process types you have scaled.
- The dyno manager (https://devcenter.heroku.com/articles/dynos) is responsible for managing dynos across all applications running on Heroku.
- Applications that use the free dyno type will sleep (https://devcenter.heroku.com/articles/free-dyno-hours) after 30 minutes of inactivity. Scaling to multiple web dynos, or a different dyno type, will avoid this.
- One-off Dynos (https://devcenter.heroku.com/articles/one-off-dynos) are temporary dynos that run with their input/output attached to your local terminal. They're loaded with your latest release.
- Each dyno gets its own ephemeral filesystem (https://devcenter.heroku.com/articles/dynos#ephemeral-filesystem) - with a fresh copy of the most recent release. It can be used as temporary scratchpad, but changes to the filesystem are not reflected to other dynos.
- Logplex (https://devcenter.heroku.com/articles/logplex) automatically collates log entries from all the running dynos of your app, as well as other components such as the routers, providing a single source of activity.
- Scaling (https://devcenter.heroku.com/articles/scaling) an application involves varying the number of dynos of each process type.

Next steps

- Perform one of the Getting Started (https://devcenter.heroku.com/start) tutorials to make the concepts in this documentation more concrete.
- Read Architecting Applications for Heroku (https://devcenter.heroku.com/articles/architecting-apps) to understand how to build scaleable apps that utilize Heroku's architecture.