O'REILLY®

Heroku Up & Running

EFFORTLESS APPLICATION DEPLOYMENT AND SCALING

Neil Middleton &
Richard Schneeman

O'REILLY®

Heroku: Up & Running

Take full advantage of Heroku's cloud-based hosting services. This guide takes you through the inner workings of this PaaS and delivers practical advice for architecting your application to work as efficiently as possible. You'll learn best practices for improving speed and throughput, solving latency issues, locating and fixing problems if your application goes down, and ensuring your deployments go smoothly.

By covering everything from basic concepts and primary components to add-on services and advanced topics such as buildpacks, this book helps you effectively deploy and manage your application with Heroku.

- Learn your way around Heroku with the command line interface
- Discover several methods for scaling your application to increase throughput
- Speed up response time through performance optimizations
- Solve latency issues by deploying your Heroku instance in new regions
- Choose the right plan for using Heroku's PostgreSQL database-as-a-service
- Get a checklist of items to consider when deploying your application
- Find and fix problems during deployment, at runtime, and when your application goes down
- Understand how Heroku buildpacks work, and learn how to customize your own

"Neil and Richard have loaded this little book with all the information you need for managing your Heroku applications."

—John Beynon
Rails developer

Neil Middleton works at Heroku writing Ruby applications. He's been developing web applications for more than 15 years across a variety of industries and technologies.

Richard Schneeman works for Heroku on the Ruby Task Force and maintains Heroku's Ruby buildpack. He teaches Ruby at the University of Texas.

SYSTEM ADMINISTRATION / CLOUD PROGRAMMING

US $9.99 CAN $10.99
ISBN: 978-1-449-34139-8

Twitter: @oreillymedia
facebook.com/oreilly

Heroku: Up and Running

Neil Middleton and Richard Schneeman

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Heroku: Up and Running

by Neil Middleton and Richard Schneeman

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Rachel Roumeliotis
Production Editor: Kara Ebrahim
Copyeditor: Jasmine Kwityn
Proofreader: Becca Freed

Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

November 2013: First Edition

Revision History for the First Edition:

2013-11-06: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449341398* for release details.

ISBN: 978-1-449-34139-8

[LSI]

Table of Contents

Preface

So, What Is Heroku?

If you're old enough and lucky enough to have had access to a computer in the early 1990s, you may have experienced the joy and wonder that was an MS-DOS game. These games were marvels of their time, but a number of them required you to do something odd, which was reboot your computer and boot into the game itself, thus taking the operating system (OS) out of the equation while the game was running.

The reason was that the OS put constraints on the game program; it used up resources needed by the game and forced certain rules into place that weren't necessarily helpful for the game to run efficiently.

As time moved on, this approach was no longer needed. Operating systems were becoming more advanced and games were able to run quite happily within their boundaries. The games were able to benefit from the additional help that the OS gave for interfacing with hardware and so on.

These days, we would never imagine running our games outside of the platform provided by the OS; it's just too much work and involvement for the developer. Why would a developer rewrite a whole hardware interface or library of sound drivers when it can rely on those provided by the OS?

Now think about this in the context of the Web. In the early days of deployment, everything was very much a homegrown affair: developers and system administrators had to figure out a lot of how things bolted together themselves, and worry about the things they had missed. As time rolled on, more and more tools and software became available, which made their lives significantly easier.

These days, though, we have platforms such as Heroku, which you could almost consider to be one of the Web's operating systems. As a developer, you can write your application in the way a game designer would—you don't need to worry about the details of how your database server is set up and you don't need to worry about how your servers are

kept up to date. Heroku is a platform that takes care of all of these things and allows you to integrate with it how you will.

Now, we are in a position where the idea of a developer building a server to host an application is becoming almost a bizarre route to take. Each day, hundreds of developers deploy applications, and fewer and fewer are getting involved in the nitty-gritty of infrastructure, instead choosing to use platforms such as Heroku.

Who This Book Is For

This book is aimed at anyone who is already using Heroku in one form or another, and wants to learn more about the best ways to make use of the technology available. This book makes the assumption that you are already proficient in using your own language of choice and are happy using Git for your source control needs.

The book assumes no previous knowledge of Heroku itself, but you will get more from it if you have deployed an application or two to the platform already.

Lastly, you should not be afraid of documentation. There is a vast array of content available, both on the subject of Heroku itself and the various languages that can be deployed to it. Therefore, instead of repeating that content, we encourage you, the reader, to go out and review whatever you can find on a given topic.

The History of Heroku

Heroku is still a relatively young company, having only started in 2007 (making Heroku younger than the Apple iPhone!). Back then, the three founders—James Lindenbaum, Adam Wiggins, and Orion Henry—were all working together at a small web development agency, and found that the amount of time spent deploying an application once having built it was not proportional. For instance, they were commonly finding that an application might take a month to develop, but then they would need to spend a week deploying the application to the Web.

After a while, they started developing applications using Ruby on Rails. With this came a newfound productivity, yet the time for deployment hadn't changed. Therefore, the three of them had the idea of bringing the same sort of efficiencies to application hosting.

In six weeks, an idea was developed and a prototype was built: a simple platform that was easy to deploy to and straightforward for a developer to figure out.

Interestingly, the initial prototype was nothing like the Heroku you see today. For instance, one major component was an application called Heroku Garden (although it wasn't known by this name at the time), a web-based code editor that users could log into and use to edit their code via a browser (see Figure P-1). Once you hit Save, the code was deployed and running on the Web, ready for users to see. Interestingly, there

are more web-based code editors popping up in 2013, which shows how far ahead of the curve it was.

Figure P-1. Heroku Garden

Over time, however, Heroku found that more and more of its target users were interested in the application hosting aspect of the product instead of the web-based editor. Therefore, Heroku developed an API that developers could use alongside Git, the source control system. Developers could also use the API to push their code to the Heroku platform.

Use of this API grew, and Heroku took the decision to turn off Heroku Garden completely. In January 2009, Heroku relaunched and rebranded itself as a Ruby/Rack deployment platform, and interest in and adoption of the product grew significantly.

Heroku is constantly developing the stack that is available for use, starting with the Aspen and Bamboo stacks, and moving to the current Cedar stack, which this book talks about.

Cedar is a big move forward in that it allows Heroku to become a fully polyglot platform (i.e., it can run many different types of applications within the same stack). Ruby is no longer the only option for use on Heroku, with the platform now supporting Python,

Java, Node.js, Clojure, and Scala, and providing the potential for a vast amount more via the use of Heroku buildpacks, which we will talk about more later in the book.

The Heroku Culture

It's probably worth mentioning the Heroku culture and how this affects its approach to software and how things are put together.

Picture the movie industry just 70 years ago. If you wanted to watch a film, it had to be what was playing at the local cinema. Hundreds of people would congregate and watch that film together, and it would be fun.

By the 1980s, the magical idea of the videocassette was taking off. Now people could choose what film to watch and when, assuming that they had access to the media. The technology was much smaller, and the media much more readily available.

Come the 1990s, the DVD made its entrance. Again, smaller devices, smaller media, but with another iteration in quality and capability. Over the matter of a couple of iterations, we had gone from a massive cinema to a box no bigger than a large book.

And these days, we've advanced yet another step. We can now stream any film we like, whenever we want, regardless of whether we own it or not. What's more, we don't need any specific hardware or media in the house. We just need a device connected to the Web.

But why are we talking about this? Well, this history matches the ethos at work within Heroku. Heroku strives to turn the age-old and complex process of hosting an application on the Web into one that requires no hardware and no software—you push your code up and the job is done. Heroku is striving to create the Netflix equivalent of application provisioning. In the world of Heroku, if there's a choice between software or service, service wins every time.

Why Would I Want to Learn More About Heroku?

By knowing more about the inner workings of the Heroku stack, you'll be able to make better educated guesses about how to architect your applications so they can work as efficiently as possible. You'll be able to identify where you need to focus your own efforts when developing your code, and which parts of your code can be left to add-ons and so on that might be available to you. You'll be able to recover an application should it go down because you will know where and what to look for. What's more, you'll understand the benefits that the platform can give you and how to encourage those around you to use the platform.

Conventions Used in This Book

The following typographical conventions are used in this book:

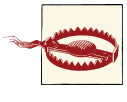
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course

Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/heroku-ur*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

From the Authors: Neil Middleton

I started to get involved with Heroku back in 2009 when the development shop I was working at was looking for a simple, effective, and relatively cheap platform for hosting our Ruby on Rails applications. After trying a few different options, we were looking at using a virtual private server (VPS) or Heroku. A few days later, we knew Heroku was the way to go. We could get applications running with the least amount of hassle in the least amount of time, and what's more, only pay for what we needed when we needed it.

As time went on, my relationship with Heroku grew closer as I got more and more involved in the platform and I started to base my website articles on best practices and know-how surrounding the product.

Eventually, a couple of things happened. First, I was approached by Heroku and O'Reilly to put this book together with Richard, and second, I ended up working at Heroku full time as part of the support team. The last year, therefore, has been a pretty dramatic

one, and one that I feel will open up some new horizons—some that will improve customer knowledge of the platform, but also to help Heroku confirm that step into the big league of application hosting.

From the Authors: Richard Schneeman

I remember vividly the first time I heard about Heroku: I had just struggled over the course of seven painful days to deploy my first Rails app to a shared hosting provider. I was running Rails 1.2 then. I looked at Heroku, saw it was free to try, and gave it a shot. I was blown away. It took a few minutes to do what took days on another provider—I was addicted. It wasn't entirely love at first deploy though; I tried several other platforms, all of which are now out of business, to see how they fared. I actually ended up using one called mor.ph for a few months because it allowed me to do things that Heroku wouldn't. Eventually, I moved on to my own VPS with a respected cloud provider. I was running a half-dozen sites on one VPS and used another dedicated VPS running a mail server to receive incoming mail for an email anonymizing service I used to run called whyspam.me.

Life was good. Or at least it was until the first of many times my sites went down. I was at work, and couldn't SSH into my box to diagnose until I got home. The fix was simple, but the message was clear: if I was going to run anything serious, I needed to get some more firepower. I switched back to using Heroku and found that those things I couldn't do once, I had no desire to do anymore. They were the same things that caused my VPS to be unstable, so I ended up rewriting those features anyway. What once seemed like arbitrary limitations were now revealed as well-seasoned advice from an opinionated platform.

This was about the same time I began to teach Rails at the University of Texas, and I needed the easiest way for my students to deploy: I chose Heroku. After the lecture on deployment, one of my students actually asked me why programming Rails couldn't be as easy as deploying it. I believe my reply started off "back in my day…"—if only they knew.

In the meantime, I began working full time for a social network called Gowalla. I loved working on it like only a crazed programmer bent to change the world can love something. When Gowalla was purchased by Facebook in 2011, I sent a note out to my favorite Ruby-backed companies. Luckily for me, Heroku—my number one choice—decided to write back, and I've been working there ever since.

I love working on Heroku, and I love introducing people to the beauty on the surface and under the covers of the platform. When the opportunity to write a few words about Heroku came up, I jumped on the chance, and here I am now.

Disclaimer

These days, things are moving fast, and especially so in technology. In fact, almost every day we are seeing changes being made to the Heroku platform. Most of these changes are invisible to the user, but changes nonetheless.

In the print world, however, things are set in stone (or ink). Therefore, there are likely to be some inaccuracies or apparent mistakes due to these potential changes. If in doubt, check out the Heroku DevCenter for the authoritative source of up-to-date documentation.

CHAPTER 1

Getting Started with Heroku

All day, every day, the Internet gets larger and larger, and recent estimates show simply phenomenal growth. Every minute, the Web is growing at a rate of over 500 new sites, while popular video hosting service YouTube is said to be receiving over 48 hours of new video content. Twitter users create over 100,000 new tweets, while Facebook users share over 600,000 new posts.

To the common observer, these numbers are amazing. However, to the developer, these numbers are simply incredible, not only because of the sheer size of the applications required to support this growth, but also the infrastructure in the form of servers and bandwidth required to keep these services ticking over happily 24/7.

The Dawn of Virtual Servers

No more than 10 years ago, the way a developer such as yourself would bring a new website or application onto the Web was to go out and purchase a new server, configure it, and find a data center that would host the server in a rack for you. This data center would provide your server with the life support that it needed to carry out its task: bandwidth, power, cooling, and the list goes on. Only once you had all these items lined up would users on the Web be able to type in your URL and find themselves looking at your website.

Now fast-forward a few years. Developers increasingly have the option of virtualization. Virtualization is the practice of renting a "virtual" server from a third party (be it another company or an internal IT team) and using that in place of your own physical machine. A virtual server would appear to the outside world to be just like a regular physical server; however, it only exists with the processes of a parent server, sharing resources and saving costs.

As a physical server is able to contain multiple virtual servers, an industry sprung up where companies would create data centers filled with machines that were sliced up and

rented out to developers to use for their applications. This brought great gains for the developer: there was no longer a need to tie up capital in purchasing a physical server; there was no need to worry about maintaining the machine if things went wrong; and what's more, it was available within hours rather than the days that it would have taken to procure and install a physical server.

Enter the Cloud

The first decade of this century brought the concept of cloud computing. As demand for hosting grew and grew, and applications became larger and larger, a new form of hosting became apparent: cloud computing. Pioneered at a massive scale by Amazon's Elastic Compute Cloud (EC2), cloud computing allowed developers to "spin up" virtual instances of servers on the Web and interact with them in the same way as normal machines. The key difference between these servers and traditional virtualization is that they are generally billed on a usage basis (usually by the hour), and at no point does the developer have any real idea of where the server is physically.

Cloud computing vendors such as Amazon do not rent out virtual servers; rather, they rent out a certain amount of computing capacity. It is up to the vendor where this capacity is provided from, and it is up to the vendor to provide and manage all the ancillary services that surround it.

Because the vendor is in complete control of the capacity and the ancillary services, new possibilities are available. For instance, suppose your website is mentioned on the Hacker News home page and you have 10 times the normal amount of traffic visiting your site. In the good old days of physical servers, your site would go down under the load and you'd be powerless to stop it, as you would be reliant on being able to purchase new hardware and configuring and installing it in your data center. With virtualization, you'd be better off: you'd be able to buy more servers from your vendor and have them up and running within a couple of hours, but until then your site would be down. With modern cloud-based hosts, you'd simply add more capacity to your application and instantly scale to meet the demand. What's more, you wouldn't be locked into this larger infrastructure, as you would with the other options—you simply scale up or down as necessary and only pay for the resources you have used.

You New Around Here?

Never used Heroku before? Jump on over to *http://www.heroku.com* and give it a spin. We won't duplicate all of the site's getting started material. Instead, we'll focus on giving you in-depth insight into how Heroku works and how to get the most out of the platform.

If you've deployed to or shared VPS servers, there are some differences (which we'll cover later), but here is a quick list to check out before getting started:

Ephemeral file system
: You can write to and read from disk, but as soon as your server restarts—and it will—that's all gone. Instead, use a shared file-storage system such as Amazon Simple Storage Service (Amazon S3). This also makes running on multiple machines easier.

Shared state
: If you want to store session data on your server, you'll need to find a way to persist it across multiple machines if you want to scale out. To do this, you can use secure cookies and a distributed store such as Memcached.

Dependency management
: If you want to install external code libraries for your app, you'll need to do it using a dependency management tool like Bundler for Ruby or Ivy for Java.

Scale out, not up
: Heroku currently offers only one server size (called a dyno); if you need more horsepower, use more dynos. If one dyno isn't big enough to get your app to run, you should consider splitting your app into smaller services, all talking over HTTP. It works for companies like Google, Facebook, and even Heroku—maybe it can work for you.

Logs
: Once you get your app on Heroku, you might need to debug your application code by looking at the logs. Because Heroku is different from a VPS or a shared host, you can't SSH or FTP into your box to see your logs. Instead, use the Heroku command line interface to run the following:

```
$ heroku logs --tail
```

: This saves you from having to SSH into multiple machines at the same time. See "The Logplex" on page 16 for more information.

Unlike anything else you've probably used before, Heroku is a platform-as-a-service (PaaS) that has plenty of opinions on how you should run your code. Although these opinions may at first seem severe, you'll get a flexible, scalable, fault-tolerant app that is a pleasure to run. If you don't want to stick to the rules, your app will not be able to run on Heroku, and it probably won't run *well* anywhere else.

Quick Start

So, at this point, if you haven't already, it is probably worth quickly playing with Heroku so that you can get an application up and running. By doing this, you can get a quick overview on the deploy and build process, plus actually get an application out there on the Web in no time at all!

The first step, if you haven't done this already, is to sign up for a Heroku account. Note that nothing we are going to do here will cost you anything, so you don't need to worry about credit cards or charges for this exercise.

Once you've got an account, make sure you've got the Heroku Toolbelt installed. This toolbelt will make sure that you've installed everything necessary for getting an application up and running on Heroku.

Once installed, log in via your toolbelt at the command line:

```
$ heroku login
Enter your Heroku credentials.
Email: adam@example.com
Password:
Could not find an existing public key.
Would you like to generate one? [Yn]
Generating new SSH public key.
Uploading ssh public key /Users/adam/.ssh/id_rsa.pub
```

For the purposes of this exercise, we'll be deploying a sample application that we've already put together, the code for which can be found on GitHub.

To get started, we need to clone this code to our local machine:

```
$ git clone https://github.com/neilmiddleton/ruby-sample
$ cd ruby-sample
```

Now that we've got the code, we can create an application to contain it:

```
$ heroku create
Creating blazing-galaxy-997... done, stack is cedar
http://blazing-galaxy-997.herokuapp.com/ | git@heroku.com:blazing-galaxy-997.git
Git remote heroku added
```

This creates the application on Heroku ready and waiting for our code, and also attaches a *git remote* to our local codebase.

Now we can deploy:

```
$ git push heroku master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 660 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)

-----> Ruby/Rack app detected
-----> Using Ruby version: ruby-2.0.0
-----> Installing dependencies using Bundler version 1.3.2
       Running: bundle install --without development:test --path vendor/bundle
--binstubs vendor/bundle/bin --deployment
       Fetching gem metadata from https://rubygems.org/.........
       Fetching gem metadata from https://rubygems.org/..
       Installing rack (1.2.2)
```

```
       Installing tilt (1.3)
       Installing sinatra (1.1.0)
       Using bundler (1.3.2)
       Your bundle is complete! It was installed into ./vendor/bundle
       Cleaning up the bundler cache.
-----> Discovering process types
       Procfile declares types     -> web
       Default types for Ruby/Rack -> console, rake
-----> Compiled slug size: 25.1MB
-----> Launching... done, v3
       http://blazing-galaxy-997.herokuapp.com deployed to Heroku

To git@heroku.com:blazing-galaxy-997.git
 * [new branch]      master -> master
```

This has taken our code, pushed it to Heroku, identified it, and run a build process against it, making it ready for deployment.

Now our application is live on the Internet! To verify this, open it now:

```
$ heroku open
Opening blazing-galaxy-997... done
```

Simple, huh?

Now that we've finished in the glory of our genius, read on to find out more about how Heroku works in the next chapter.

CHAPTER 2

How Heroku Works

At the time of writing, Heroku has had over three million separate applications deployed on its infrastructure, and this number grows day by day. Running a large number of applications day to day requires a substantially different approach than running just a handful, and this is one of the reasons that the Heroku architecture is markedly different from what you or I might develop if we were setting up our own environment on our own hardware for a single application. Heroku's task is to support the running of all of these applications at the same time, managing the deployments that users are requesting, as well as scaling applications' needs.

In order to achieve this, the Heroku platform is broken up into several key segments; the most important among these are:

- *Routers*, which ensure your application receives the requests from users on the Web
- *Dynos*, where your application code actually runs day to day

In addition to these, there are a number of extra components, such as the Logplex and the various add-ons and services available to your applications (see Figure 2-1).

So, let's walk through all of these parts in turn.

What Is a Dyno?

When you deploy an application to Heroku, it is run in a container called a *dyno*. The more dynos your app has, the more running instances of your application are available to take requests. Each dyno is completely isolated from other dynos. You can add dynos to extend capacity and to allow for fault tolerance (since one dyno going down will not affect other dynos). At the time of writing, each dyno represents 512 MB of physical RAM.

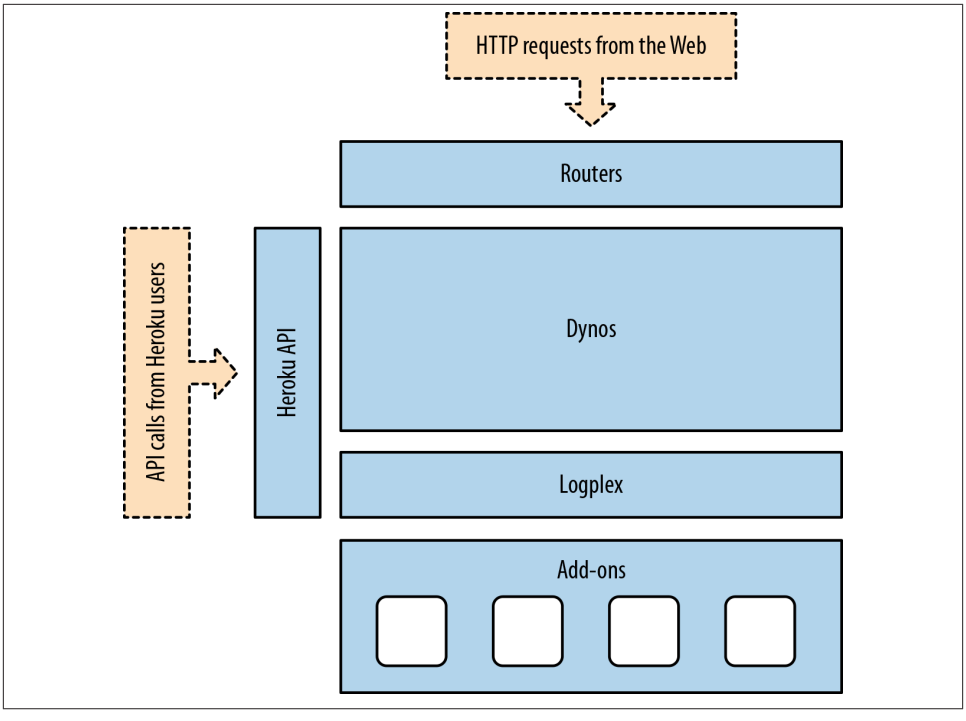


Figure 2-1. Overall architecture of the Heroku platform

Applications are billed by the dyno/hour prorated to the second, and every application is credited with 750 dyno/hours per month. The caveat here is that Heroku idles any applications running a single dyno when not in use to free resources for other applications.

So, a dyno is like a small virtual computer, but if your application is split up over multiple compute units, how does Heroku know which dynos to send requests to?

HTTP Routing

When a user types your application's URL into a browser or tries to make a request to it via an API (and so on), there needs to be a way to connect the user with your application running somewhere deep down within the Heroku platform. There are thousands of applications running on Heroku, and maintaining a persistent location for an application within the platform is an approach fraught with danger.

The Heroku router is a piece of software (written using Erlang, called Hermes), which acts as a gateway between your users and the dynos that are running your code. In its memory, it is aware of every single application currently deployed to the platform and the external URLs that each of these applications should respond to (including the

.herokuapp.com URLs, legacy *.heroku.com* URLs, and any custom domains that you may have added). Lastly, it stores the current location of your application on the platform at the current time.

Therefore, the routers have a lot to keep track of. So what exactly does it look like when a request comes in from a user on a web browser?

Request Life Cycle

When a user wants to visit your site, he types the URL into the address bar and hits Enter. Then a DNS query is made to your provider. They see you've pointed your address at Heroku either using an A record or a CNAME. With this information, the request is sent to the Heroku routers.

Every time a router receives a request, it carries out a lookup on the URL being requested and determines the location of your application code. Once found, it fires the request at your code and awaits a response. This is where your application steps in and handles the request, doing whatever you programmed it to do.

Once your code has completed processing the request and a response has been returned, the router will pass the response back to the end user. The best part about this is that the routers are completely transparent to the outside world.

Now that you know what happens when a request goes through, what happens if something hangs in your application? Does Heroku just keep thousands of dead requests alive?

Long-Running Requests

As a way of protecting the user from long-running requests, the router processing the request will wait for only 30 seconds before returning a timeout error to the end user. The error returned in these instances shows in your application logs as an H12. Note, though, that this only counts the first byte returned. Once that first byte of response is returned, the router sets a 55-second rolling window before an error is returned (in these instances, the error code changes to an H15). This, therefore, means that you are effectively able to stream responses back to the user without worrying about hitting this timeout error.

Let's say we're sending a valid request that our server should be able to respond to. We know that our code lives on a dyno, but where exactly does a dyno live?

The Dyno Manager and Dynos

In simple terms, a dyno is an isolated, virtualized UNIX container that provides the environment required to run an application.

Each dyno running on the platform will be running different application code (see Figure 2-2). For instance, if it were a Ruby on Rails application, one dyno could be running an instance of Unicorn, whereas if it were a Java application you might see Tomcat or something similar. Across this array of dynos, you may have potentially thousands of applications, with each dyno running something different. Some applications running at a higher scale will be running on more than one of those dynos, some will be running on only one or two, but the key thing to remember here is that a single application could be represented *anywhere* within this system.

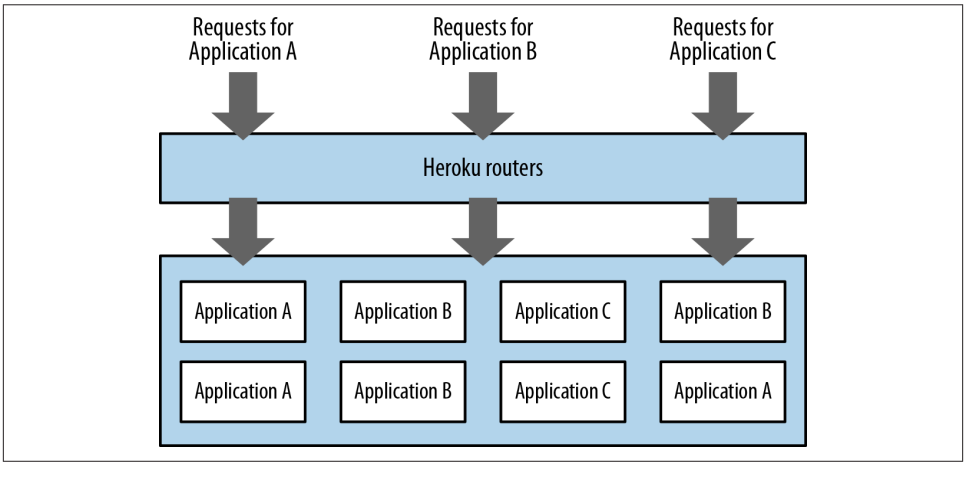


Figure 2-2. The Dyno manager

Before your application code is paired with a dyno, the dynos are all identical. This uniformity allows Heroku to easily manage the thousands of applications on the platform. When you send your code to Heroku it is compiled into a format known internally as a "slug" that can then be quickly run on any of the available dynos. Because Heroku manages such a staggering array of dynos, it is inevitable that performance of an underlying server may slow to a halt or a hardware failure will occur. Luckily, when this is detected, dynos will be automatically stopped and restarted.

While 10,000 hours of uptime on a hard drive might sound like a long time, if you're running 10,000 hard drives, that's one failure per hour.

Because of this, Heroku advocates building stateless web apps. As an additional measure toward stability, each dyno is restarted once every 24 hours. All of this requires no interaction on your part, freeing you up to focus on developing your app code.

Configuration

A given codebase may have numerous deployments: a production site, a staging site, and any number of local environments maintained by each developer. An open source app may have hundreds or thousands of deployments.

Although all running the same code, each of these deploys have environment-specific configurations. One example would be credentials for an external service, such as Amazon S3. Developers may share one S3 account, while the staging and production sites each have their own keys.

The traditional approach for handling such config vars is to put them under source (in a properties file of some sort). This is an error-prone process, and is especially complicated for open source apps, which often have to maintain separate (and private) branches with app-specific configurations.

A better solution is to use environment variables, and keep the keys out of the code. On a traditional host or working locally, you can set environment vars in your `bashrc`. On Heroku, you use config vars:

```
$ heroku config:set GITHUB_USERNAME=joesmith
Adding config vars and restarting myapp... done, v12
GITHUB_USERNAME: joesmith

$ heroku config
GITHUB_USERNAME: joesmith
OTHER_VAR:       production

$ heroku config:get GITHUB_USERNAME
joesmith

$ heroku config:unset GITHUB_USERNAME
Unsetting GITHUB_USERNAME and restarting myapp... done, v13
```

Heroku manifests these config vars as environment variables to the application. These environment variables are persistent (they will remain in place across deploys and app restarts), so unless you need to change values, you need to set them only once.

By making use of these configuration variables, it is therefore possible to have two different dynos containing the same application code that behave differently. For instance, one may be a production grade application that sends emails to its users, whereas the other may be a development system that emails only the developers.

So a dyno is a container that holds our code and our configuration, but how do we get our application code into them? Heroku calls this process *slug compilation*.

Releases

Before we talk about the process of creating a release that is deployed onto Heroku, let's quickly define what a release is.

As far as Heroku is concerned, a release comprises a combination of your application code (or slug), and the configuration around it. Therefore, any changes to either of these will generate a new "release," which can be seen via the `releases` command:

```
$ heroku releases -a neilmiddleton
=== neilmiddleton Releases
v62  Deploy e5b55f5                neil@heroku.com        2013/04/29 22:24:44
v61  Deploy 5155279                neil@heroku.com        2013/04/24 18:44:57
v60  Add-on add newrelic:standard  neil@heroku.com        2013/04/19 00:04:06
v59  Add papertrail:choklad add-on neil@heroku.com        2013/04/19 00:03:00
v58  Deploy 0685e10                neil@heroku.com        2013/04/18 17:53:20
v57  Deploy 823fbdf                neil@heroku.com        2013/04/18 17:25:55
v56  Remove librato:dev add-on     neil@heroku.com        2013/04/16 23:42:39
```

This definition of releases makes it possible to roll back to previous versions of your code and the configuration surrounding it. This is very useful if a deploy goes bad or your configuration changes cause unforeseen issues.

Slug Compilation

From the moment Heroku detects you are pushing code, it will fire up a runtime instance to *compile* your app (in the very early days of Heroku, this wasn't needed, as Rails was the only supported web framework). Additionally, external dependencies, such as gems, could simply be preinstalled on every dyno. Although that was a neat idea, the list of external dependencies your code might rely on grows exponentially. Instead of trying to preinstall all of that software, Heroku relies on tools adopted by the community for dependency management.

Before any code is coupled to your dyno or any dependencies are installed, the dyno has a base image of preexisting software. This is known internally as the *runtime* because the image will need to have enough functionality to run your code or at least install dependencies needed to run your code. This image is kept fairly lightweight to minimize overhead. All software on the base image (e.g., cURL) is available as open source. This is an intentional choice to maximize code portability and to increase compatibility with a user's development machine. This includes the base operating system, which is and always has been a flavor of Linux. By keeping all the base images of dynos the same, security updates of components are completed by Heroku engineers quickly and with little or no impact to running applications.

Although some criticisms of platform as a service (PaaS) involve the so-called "vendor lock-in," such incompatibilities in the ecosystem would indicate a cost associated to adopting the platform. It is Heroku's goal to provide as seamless and transparent a deploy

process as possible while maintaining reliability and consistency. Heroku supports open source software and so should you.

So, now that Heroku has a secure base image with suitable tools, and it has your application code, it runs a buildpack to determine exactly what needs to be done to set up your code to run. The buildpacks are all open source and can be forked and customized. For more information on how buildpacks work, see Chapter 8.

Once the buildpack is done executing successfully, a snapshot of the finished product (minus runtime image) is placed in storage for easy access at a later time. This product is referred to as a *slug*, and you may have noticed while deploying that Heroku will tell you your slug size. One reason to pay attention is that larger slugs take longer to transfer over the network, and therefore take longer to spin up on new dynos.

Now that your slug is saved, dynos carrying old application code are sent commands to begin killing off their processes. While this is being done, your code is copied over to other dynos. Once the old application quits running, the new dynos are brought online and made available to the routers ready to serve requests.

Because Heroku keeps a copy of your application in storage, if you need to scale out to more dynos it can easily copy the slug to a new dyno and spin it up. This type of quick turnaround means that you can do things like run a scheduler in its own dyno, and every time a command is executed on one of your apps via the `run` command, like `heroku run bash`, you are actually running it inside of a completely fresh and isolated dyno. This protects you from accidentally running `rm -rf` on your production web server.

Now that you know a bit more about what actually goes on in the process of building a slug and spinning it up as a running process, let's take a look at how we can use this to our advantage.

Scale Out, Not Up

Traditionally, when servers run out of capacity, they are scaled up (i.e., developers turn off the machine and add RAM, a bigger hard drive, more cores, a better networking card, etc., and then turn it back on). This might buy some time, but eventually as an application grows in user base, a server will max out on upgrades and engineers will have no choice but to scale out. Scaling out is when you add extra servers for capacity rather than extra capacity to your server. This is a great practice because you can scale out in an unlimited way, whereas scaling up is very limited. Typically, scaling up is easy, and scaling out is hard. It requires provisioning new hardware, networking it, installing software, patching and updating that software, building out a load-balancing infrastructure, and then buying a bunch of pagers since more hardware means more failures.

When you run on Heroku, your application is already primed to scale out, all without the pagers. Storing *compiled* application code into slugs and keeping this separate from

your running instances gives you a massive amount of *elasticity*. When you give Heroku the command, it pairs your compiled slug with a dyno to give you more capacity. This means you could take your app from 2 dynos to 102 dynos with one command (that's 51 gigs of RAM if you're doing the math at home). So, if your big V2 product launch is coming up in a few days, you can sleep soundly knowing that extra capacity is available if you need it, but you're not obligated to use it.

This ability to scale out is not by accident, and it comes with a host of side benefits. As we mentioned earlier, it makes your application more fault tolerant of hardware errors. For instance, a single server could be running several dynos when the memory fails. Normally this would be a significant issue in traditional environments, but the dyno manager is able to spot the dead instance and migrate the dynos somewhere else in a heartbeat (see Figure 2-3).

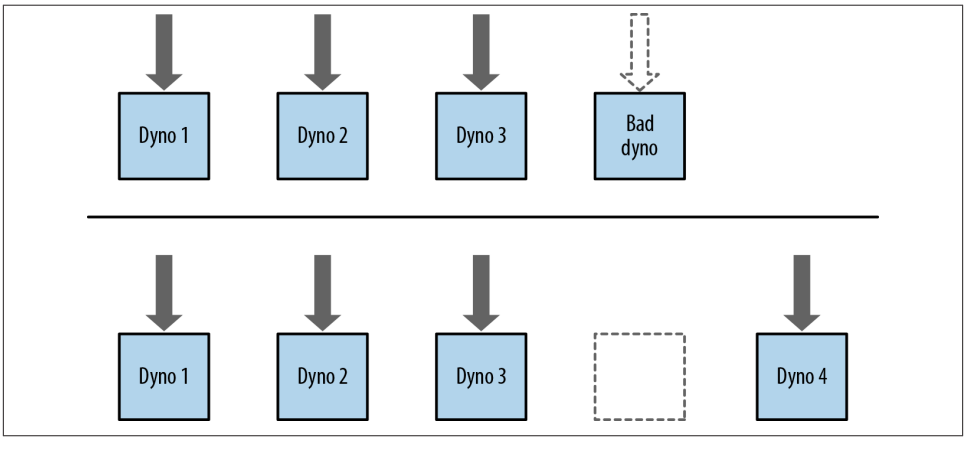


Figure 2-3. The dyno manager recovering from a node failure

Another benefit of the dyno architecture is isolation. Each dyno is a completely separate container from all others, so there is no possibility of another application accessing your application code, or suffering because a rogue dyno is spinning out of control and using up all the CPU on the host instance. As far as you the developer are concerned, these are all separate entities.

Erosion Resistance

A nontechnical benefit to the dyno architecture is *erosion resistance*. Software erosion is what happens to your app without your knowledge or consent over time: it was working at one point but doesn't work anymore. Imagine a security vulnerability is discovered in the operating system your app is running. If you don't patch the vulnerability quickly, attackers may take advantage of the exploit and gain access to your code,

or worse. Although you didn't intentionally put the vulnerability in your app, doing nothing erodes the ability of your app to perform its job safely and securely.

When your application experiences an error due to an erosion of software, you have to invest energy diagnosing and resolving the problem. Erosion is a problem that all applications suffer from regardless of size, and one that many applications have problems dealing with if left for too long.

Software erosion can be thought of principally as your application getting "out of date." Your application consists of many moving parts, some of which are being constantly updated by the communities around the world. For instance, you have operating system upgrades, kernel patches, and infrastructure software (e.g., Apache, MySQL, SSH, OpenSSL) updates to fix security vulnerabilities or add features. All of these need to be kept up to date and are often left by the wayside due to the effort required to do so.

Because all dynos are derived from the same image, it is guaranteed to be the latest and greatest image that the Heroku operations team has put together at the time the dyno is created. These images are put through lengthy test processes and evaluation before being placed into production.

Once your application is up and running, its dynos are silently *cycled* automatically once a day to ensure that the dyno image version they are running is the latest available runtime (your slug will stay the same). You can check your application logs to see that the cycling process has occurred. Therefore, if you were to deploy an application to Heroku and leave it for several months, it would be no more out of date than one deployed a day ago.

Workers and Background Processing

Not all dynos need be ones that respond to HTTP requests. A dyno can be thought of as a container for UNIX processes. Dynos can therefore handle a variety of different types of tasks. While your web dynos will handle all your web requests, other dynos may be tasked with carrying out some sort of background processing, be it working a job queue, sending emails, or managing event triggers based on a calendar. The list of possibilities here is endless. However, note that these dynos still have an attached cost and will therefore need consideration when planning your application architecture and projecting costs.

A very common use case of a worker dyno is to process outgoing email. When a user signs up on a website, she will typically get a confirmation or welcome email. If you are sending this message during your web request, your user must wait for the email transaction to complete before she gets to go to the next page. This makes your site seem slow and can actually lead to timeouts. By putting that information into a lightweight queue and sending that email off from a background worker, you can improve the speed of your web server while increasing its capacity.

You can also schedule one-off background tasks with Heroku's scheduler. This is like a worker, but only runs a specific command for a given interval. This is very similar to how you might schedule a task to be run with *cron*. Tasks run by the scheduler are only charged for the time they use, so if a task takes only 12 minutes to complete, your account will be debited only 12 dyno/minutes.

Now that we've got our web stack and our worker infrastructure under control, let's take a look at the other features Heroku provides to help developers.

Other Services

We've discussed the main components of the Heroku platform that keep your application code up and running, so now let's talk about some of the supporting services that make up the parts of the platform the user doesn't see. These include the Logplex, deployment systems, and the various add-ons that you are able to attach to your application to provide ancillary services.

The Logplex

Every application generates output, be it content via generated web pages or data via an external API. Another form of output that applications create are logfiles on the server. If you've run a production web server, you'll know there is no substitute for easy access to logs.

In a traditional web server configuration, many logfiles are created. First, you have the regular system logfiles that are generated by the operating system itself. Then you have the logfiles created by the services running on the server, such as Apache (web server) logs, or PostgreSQL (database server) logs. Lastly, you have your own application logfiles.

Typically, all of these files are written to the local disk on their respective servers before processes such as UNIX's *logrotate* come along and archive the files to another location, usually compressing them down. A respectable system administrator at this point will periodically archive these logfiles to a different location for later interrogation or reference.

One of the downsides of logfiles is that they affect the way we look at the data within a system. Application and system logs contain a time-ordered stream of information about what is happening in your application at any given time. Although this stream is constant and ongoing, the way they are typically represented in logfiles tends to imply that there is a beginning and an end. To address this misperception, Heroku developed the Logplex.

The Logplex is an open source tool that replaces the logfile concept by providing a constant stream of information flowing from your application with no beginning and

no end. What's more, this stream contains a single canonical source of information from every part of the stack—be it the routers, your dynos, and so on. This means that as a developer you are able to tap into this feed at any time and see all of your application's activity in one place at the same time. Therefore, Heroku does not provide logfiles per se, but offers this stream. Should you wish to capture this information, Heroku allows you to *drain* this information into an external store.

Here's a sample of Logplex output from a simple web application:

```
Jul 19 19:37:30 MyApplication_Production heroku/router:  GET
  www.myapplication.com/admin/applicants dyno=web.1 queue=0 wait=0ms
  service=381ms status=200 bytes=22849
Jul 19 19:37:30 MyApplication_Production app/web.1:  Completed 200
  OK in 370ms (Views: 213.0ms | ActiveRecord: 131.6ms)
Jul 19 19:38:27 MyApplication_Production app/postgres:  [5-1] postgres
  [www] LOG:  duration: 61.517 ms  statement: SELECT count(*) FROM
  pg_stat_activity;
Jul 19 19:41:17 MyApplication_Production heroku/router:  GET
  www.myapplication.com/ dyno=web.2 queue=0 wait=0ms service=14ms
  status=302 bytes=115
Jul 19 19:41:17 MyApplication_Production app/web.2:  Started GET
  "/" for 180.76.5.168 at 2012-07-19 16:41:17 +0000
Jul 19 19:41:17 MyApplication_Production app/web.2:  Processing by
  User::SessionsController#index as HTML
Jul 19 19:41:17 MyApplication_Production app/web.2:  Completed  in 6ms
Jul 19 19:41:57 MyApplication_Production app/postgres:  [19753-1]
  [www] LOG:  checkpoint starting: time
Jul 19 19:41:58 MyApplication_Production app/postgres:  [19754-1]
  [www] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0
  transaction log file(s) added, 0 removed, 1 recycled; write=0.000 s,
  sync=0.000 s, total=0.014 s; sync files=0, longest=0.000 s,
  average=0.000 s
Jul 19 19:45:38 MyApplication_Production app/postgres:  [5-1] postgres
  [www] LOG:  duration: 61.340 ms  statement: SELECT oid, typname
  FROM pg_type where typtype = 'b' AND typname IN ('hstore')
Jul 19 19:46:58 MyApplication_Production app/postgres:  [19755-1]
  [www] LOG:  checkpoint starting: time
```

Databases and Other Add-Ons

So far we have talked about how Heroku can take your application code and run it in the cloud, allowing you to scale to meet your application needs. However, we haven't talked about how Heroku provides all the other services that a modern web application needs.

It is not uncommon these days for an application to require more than just code. Databases are an extremely common requirement, as are external services such as Email SMTP or caching services such as Memcached. Lots of other services are becoming more common as well: Redis, AMQP, full-text search, image and video processing, NoSQL databases, and SMS are all growing in usage on a daily basis.

So how does Heroku provide these services? Well, in short, Heroku doesn't. From a very early stage, Heroku put together the Add-on program, which is a process whereby an external vendor can develop a service and then make it available to the platform via the add-ons API.

By using this approach, it has been possible for a library of add-ons to build up over time that can provide almost all the ancillary services that an application might ever need to consume. Because most of these services are attached via simple configuration, the effort required to use these services with their own application is very low.

More information on the add-ons library can be found here.

One of the most popular add-ons is one provided by Heroku itself. Called Heroku Postgres, this add-on provides PostgreSQL services to all applications.

On initial deployment, Heroku will detect the language and type of your application. If you are deploying using a popular database-backed web framework, such as Ruby on Rails, Heroku will provision and configure the PostgreSQL add-on for your application. This means that deploying a database-driven application is as simple as pushing your code to Heroku and running a migration.

The Heroku Postgres service comes in a variety of shapes and forms, so there is a dedicated chapter to this topic later on in the book (Chapter 5).

However, not everyone requires PostgreSQL; for example, you might want to use MySQL or some other form of database with your application code. Although Heroku recommends PostgreSQL, most of the alternatives can be found in the add-ons library (e.g., Amazon's Relational Database Service [RDS] or the ClearDB MySQL service). For more information on these alternative database services, see the relevant sections in the add-ons library.

Deployment Systems

One of the most commonly overlooked parts of the Heroku platform is the deployment system. This incredibly complex part of Heroku is what takes your code and transforms it into a platform-friendly version that can be scaled up and down instantly and at will. What's more, the deployment system also alters your application to suit the platform, injecting various components and configuration to make your life, as the developer, much easier and simpler.

All Heroku deployments are carried out via the use of the Git source control system. By *pushing* your code to Heroku you are asking the platform to deploy your code. But how does this work?

Well, for starters, Heroku holds a copy of the Git repository for every single application on the platform. When you push into the master branch, you initiate a deploy and Heroku will start making your repository ready for deployment. You can push into other branches if you wish, but these will only be stored on Heroku and will not be deployed, as the platform will only ever deploy the master branch. However, it is possible to push from a remote feature branch such as staging into the Heroku master. You will learn more on this later in the chapter, when we talk about application management.

When Heroku receives your code via `push`, several things occur. Heroku takes your code and identifies it (e.g., as a Rails application, Django, Java, etc.). Once identified, it runs the code through a buildpack. Buildpacks are special programs that understand the structure of your application and its appropriate conventions. By understanding your application structure, these buildpacks are able to make the changes required for your application to function correctly on the platform. This can include a variety of changes such as ensuring configuration settings are correct, injecting application framework plug-ins, or overwriting configuration files such as database configurations.

Buildpacks are provided by the Heroku operations team, as well as by other open source developers. Anyone can write their own buildpack for their own means. For more information on buildpacks in general, see Chapter 8.

Once your application has been prepared, it is stored in a slug. This slug is stored in a file store waiting for a deployment request, as mentioned earlier. In most cases, the slug is immediately deployed to new dynos, replacing the ones that you already have running.

Most use cases already have a buildpack prebuilt by the Heroku operations team. For instance, if you were to push a Django application to Heroku, the hard work of slug compilation is done for you, and there is nothing more you need to do. Should you wish to push something a little off the beaten track or if you need to customize the way that your application deployments are handled, you are now able to write your own custom buildpacks and ask the platform to use these instead of the default. More information on this can be found in Chapter 8, which goes into detail about buildpacks.

All of the default buildpacks can be seen on GitHub.

CHAPTER 3

Understanding Performance and Scale

When talking about performance in apps, two factors affect your users: speed and throughput. Speed is how fast your application can take a request and return a single response. Performance, when most people talk about it, refers to raw speed. Throughput is how many simultaneous requests can be handled at the same time. If speed is a drag racer screaming down the track, then throughput is thousands of participants crossing the line of a marathon.

Speed and throughput are closely related. If the throughput limit of your application isn't high enough, then individual users will likely see slower response times, or no response at all. Likewise, speeding up individual requests with no other changes by definition means your app can process more requests per second. It's important to account for both of these factors when considering performance. So, how can you make your app faster while handling more load at the same time? You'll just have to keep reading to find out.

Horizontal Scaling and You

In the past, if you had an application that needed to handle more concurrent connections, you could increase the RAM, upgrade the hard drive, and buy a more expensive processor. This is known as vertical scaling; it is very easy to do but is limited. Eventually, you'll max out your RAM slots or some other component. Instead of adding more capacity to a single computer, you can horizontally scale out by networking multiple computers together to form a horizontal cluster. Although this allows virtually unlimited scaling, it is difficult to set up, maintain, and develop on. Luckily for you, Heroku takes care of most of the difficult parts for you.

Each dyno on Heroku is an identical stateless container, so when your app starts getting too many requests at a time and needs more throughput, you can horizontally scale by provisioning more dynos. Heroku takes care of the networking, setup, and maintenance.

What has in the past taken hours of setup can be accomplished on Heroku with a simple command:

```
$ heroku ps:scale web=4 worker=2
```

You just need to make sure that your app is architected so that it can run on multiple machines.

Stateless Architecture

Although Heroku maintains spare capacity for your application so you can scale, you still have to build your app so that it can run across multiple machines at the same time. To do this, your app needs to be "stateless," meaning that each and every request can be served by any dyno.

In the early days of the Web, it was easy to store files and "state" as logged information, to an application's memory or to disk. This makes it much harder to horizontally scale, and can break user expectations. Imagine you sign up for a new web service, upload your picture, and click Save. The picture gets written to one of four machine drives used for this service. The next time you visit the site, your request goes to one of the three without your photo on the machine, which will make the website appear broken.

Many "modern" frameworks support and encourage stateless architecture by default. Even so, understanding the features of stateless architecture is crucial to using it correctly. There is a long list, but the two most common are using a distributed file store and managing session state. Let's take a quick look at them.

Distributed file stores

Continuing with the website example, a file written to one machine that is not available to the others is a problem, so how do you get around it? By creating a distributed file store that is shared and accessed by all the applications. The most commonly used product for this purpose is Amazon's S3. As a file, such as a user's avatar image, comes into your app, it is written to S3 and can be read from all machines. Doing this also prevents your app from running out of disk space if too many files are uploaded. It is common to keep a record of the file in a shared database between dynos so that files can be located easily. There is also an add-on called bucket that provisions a distributed file store for you.

It is inadvisable to write data directly to disk on a Heroku dyno. Not only is it not scalable, Heroku uses "ephemeral" disks that are cleared at least once every 24 hours. Most frameworks support easy integration with third-party distributed file stores, so take advantage of them. Now that you don't have to worry about file state being different on different machines, let's take a look at session state.

Session state

When a user logs in to a website, information about that user is captured (e.g., email, name, when she last logged in, etc.). It is common to store some identifying information about the user in an encrypted cookie on the user's machine. This cookie usually contains just enough information to look up the needed information. If your application is storing the related details, name, email, and so on (either on disk or in application memory), then a user will appear logged out when visiting different machines. To fix this, we need a distributed data store to keep that information so all dynos have access. It is common to store this information in a relational data store such as Postgres, or in a distributed cache store such as Memcache.

Many popular frameworks, such as Rails, store session state this way out of the box, so you don't need to worry about it. If you're using a lighter framework, or some custom code, then you should double-check that session state is safely shared between dynos.

Now that you understand why your application should be running statelessly, let's take a look at the Dyno Manifold. This is the component in Heroku's stack that allows us to easily scale our dynos.

Dyno Manifold

The Dyno Manifold always maintains spare capacity that is ready and waiting to receive new running dynos at a moment's notice. Let's say your application is new and running on a single dyno, having just been freshly deployed, and now you want to scale it up to six dynos. A simple request to the Heroku API issues commands to the platform to create five new dynos within the spare capacity and deploys your application to them. Within seconds you now have six dynos running and therefore six times the capacity that you had only a few seconds ago.

Performance versus Scaling

Let's clarify a common misconception of what scaling will do to your application. A common belief is that more servers (or in this case, dynos) causes increased *performance* for your application. This is incorrect. Scaling your application gives you an increased *concurrency* (the ability to serve more than one request at the same time).

Imagine, for a moment, an application that has a mean request time of 100 ms. In an ideal world, in any given second, this application, when serving a single request at a time, can thus serve 10 requests a second. Adding a second dyno will not change this, as your code will always take a finite time to process; only software engineering and optimization will reduce this time. This second dyno, though, does mean that your application, across the two dynos, is now able to serve twice the number of requests at a time.

Autoscaling

Autoscaling is a myth. Heroku does not autoscale your app for you, but not to worry—it's pretty simple to learn when to scale and even easier to do the scaling.

> Why doesn't Heroku just build an autoscaler? It is super easy, I have a friend who has an algorithm that…
>
> — Everyone ever

Scaling is a surprisingly complex problem. No two applications are the same. Although scaling usually means adding extra dynos for extra capacity, that's not always the case. If the bottleneck is with your database, adding extra frontend dynos would actually increase the number of requests and make things even slower. Maybe your Memcache instance is too small, or, well, you get the point really—it could be a thousand other tiny application-specific details.

One concern to be aware of is how your backing services may be crushing your ability to scale. One easily overlooked limitation of horizontal scalability is database connections. Even the largest of databases are limited in the number of connections they can accept. Provisioning additional dynos will give your application more processing power, but each of these dynos will require one or more connections. It is important to be aware of this maximum number and to begin to take appropriate preventative measures such as sharding your database or using a collective connection service such as PGBouncer.

At the end of the day, the best autoscaler is the one reading this book. You know your application better than any algorithm ever could. Heroku can give you access to the tools to measure and make scaling easier than ever, but there should be a few eyes on the process. External services can provide autoscaling behavior, but when it comes to capacity planning and performance, there is no substitute for a human. Also, if any of your friends solve this problem, they could make a good chunk of change making an add-on.

Let's take a look at how you can estimate your required resources.

Estimating Resource Requirements

By understanding your mean request time, and knowing what sort of traffic levels you receive at a particular time of day, it is possible to predict approximately how many dynos you require to suit your application needs.

For example, imagine you have a peak load of 1,500 concurrent users generating traffic at 500 requests a second. You can check your logs or use a monitoring tool such as New Relic to know that your mean request time is 75 ms. We can simply calculate the number of required dynos by working out how many requests per second a single dyno could handle (1,000 ms divided by 75 ms equals 13 requests) and how many dynos at 13 req/sec

are required to serve 500 requests (so 500 divided by 13). This gives us the final answer of 38 dynos to handle your traffic:

```
Requests Per Second / (1000 / Average Request Time in ms)
```

Note, though, that it's not quite that simple. While predication and some simple math will get you so far, some application containers, such as Unicorn and Tomcat, are able to serve multiple requests per second, so in these cases one dyno may be able to serve three or four concurrent requests alone. Traffic also fluctuates massively, so predicting a throughput for your application is generally quite difficult unless you have a wealth of historical trends for your particular application. The only sure-fire way to ascertain the required resources for your application is to bring up the approximate number of dynos you believe you require, and then monitor your application as the traffic comes in.

Request Queuing

It is worth mentioning at this point the concept of request queuing. When a request arrives, it is randomly sent to one of your applications and available dynos for processing. Should a dyno be unavailable for some reason, it is dropped from the pool of dynos receiving requests automatically. However, remember that while this is happening, the router processing the request is counting down from 30 seconds before returning an error to the user. Therefore, it is important to monitor your application's queue length to see if more resources are required to handle the load. Because of this, it can be helpful to offload non–HTTP-specific tasks to a background worker for asynchronous processing.

Speeding Up Your App

Now that you understand how to scale your app for throughput, we will take a look at a few different ways to speed up your response time. After all, everything today is measured in time, and when it comes to applications, the quicker the better.

Because speed refers to the amount of time it takes to get a response to the user, we will cover a few backend performance optimizations as well as some on the frontend. Although there are big gains to be made in the backend, a browser still takes time to render content, and to visitors of your site, it's all the same.

We'll start off with some very simple frontend or "client-side" performance optimizations and then move on to more invasive backend improvements. Let's get started with how we serve static files to your visitors.

Expires Headers

When you visit sites like Facebook and Google, you might notice large logos, lots of CSS, and a good amount of JavaScript if you were to view the source. It would almost be insane to have to redownload all of these files every time you clicked on a link or refreshed the page; instead, your browser is smart enough to cache content it knows will not change on the next page load. To help out browsers, Google's servers add a header to each static file; this lets a browser know how many seconds it can safely keep a file in cache:

```
Cache-Control: public, max-age=2592000
```

This tells the browser that the file is public and can serve the same file for the next 2,592,000 seconds. Most frameworks will allow you to set the cache control globally.

While setting an expires header far into the future can be good for decreasing repeated page loads, you need to make sure that when a static file is altered (i.e., when a color is changed from red to blue), that the cache is properly invalidated. One option is to add a query parameter to the end of the filename with the date that the file was last saved to disk. So a file called *smile.png* would be served as *smile.png?1360382555*. Another option is to take a cryptographic hash of the file to "fingerprint" it. If you were to take an MD5 hash over your *smile.png*, it might produce a "hash" of *908e25f4bf641868d8683022a5b62f54*, so the output filename would be something along the lines of *smile-908e25f4bf641868d8683022a5b62f54.png*. The key here is that the URL to the static file is different, so the browser doesn't use the old cached file.

Different frameworks support different cache invalidating strategies, so check the documentation of your favorite frameworks for more information.

Now that you know how to reduce repeated page load time, how can we reduce all page loads? One option is to use a CDN.

Faster Page Response with a CDN

Heroku's Cedar stack has no Nginx or Apache layer to serve static files. While this means more system resources on a dyno to run your application, and more control, it also means that your app has to use extra resources to serve static files or assets. Instead of using valuable dyno time to serve these files, you can serve files from an edge-caching CDN.

CDN stands for Content Delivery Network. A CDN is made up of many different machines that are spaced all across the globe and are geolocation aware. By moving your assets to a CDN, not only do you reduce load on your app, you also decrease latency of each individual file request. A CDN is likely closer to your visitor's location than your app running on Heroku. The math is pretty simple: the less distance data has to travel, the quicker it gets there. In the past, using a CDN to serve these static files meant having

to copy assets to a third-party server. Instead of copying files using an "edge cache," the CDN pulls assets straight from your application and caches them, so there is no syncing involved.

In general, you will set up a CDN to pull from your app *example.herokuapp.com* and in return you will get a dedicated CDN subdomain such as *jbn173nxy4m821.cdndomain.com*. Once in place, you can then get to your assets via that subdomain, so an image *assets/smile.png* would be located at *http://jbn173nxy4m821.cdndomain.com/assets/smile.png*. This will tell the CDN to download that image and serve it directly from the edge caches. There are various services that provide edge cache CDNs, such as CloudFront and CloudFlare, that can be manually integrated with your Heroku application.

As more and more Internet traffic goes global, your visitors depend on you to provide them with the quickest experience possible. Using a CDN in your application is one way you can help to meet their needs.

One of the most notorious causes of slow apps, as we've mentioned, is the database. Let's take a look at what we can do to speed things up there.

Postgres Database Performance

Improper use of any data store will lead to slow performance and a slow application experience for any of your visitors; Postgres is no exception. This relational database offers many ways to tweak settings and measure performance. Here we'll look at just the most common causes of slow app database performance: N+1 queries and queries in need of an index.

N+1 queries

One of the most common database problems comes not from improper setup but from accidentally making many more queries than are needed. Imagine you just started a new social network where a user can have friends, and those friends can all have posts. When a user loads her user page, you want to show all of her friends' posts. So, you grab all of the user's friends, iterate over that array to grab all of the posts from a database, and then show that result. If a user has 100 friends, and there is an average of 9 posts per friend, you just had to make $(100 \times 9) + 1 = 901$ database queries. This problem is called N+1 because we've had to make 900 queries to get the data we want plus the first query to grab our user. Even if each of these queries take a fraction of a second, they quickly add up, and the more friends a user has, the longer it will take for the page to load.

Instead of doing these calculations manually, it is much easier to tell the database about all of the data we want. Postgres is a relational data store, so we can do this using `joins` to grab the exact values we need. This common pattern is known as *eager loading* since we're using a `join` to preemptively grab the extra data we know we will eventually need.

The best way to look out for this cause of slowness is to check your application logs in development, assuming that queries to the database are logged. If you have this problem, you'll likely see hundreds of SQL queries per request where you should only see a few.

Properly using relations can help tremendously in allowing the database to do its job. However, you still may find that your application is slow; perhaps it could benefit by adding an index.

Queries in need of an index

Relational databases have been around for decades—plenty of time for optimizations and speed improvement, but it doesn't mean that they can read your mind. If you know ahead of time what columns you will be querying your table with, you can dramatically speed up that query time by using an index. When you ask a database to find a row in the users table that has a value of *foo@example.com* in the email column, the database will perform what is called a "sequential scan" to find the row you want (i.e., it looks at each field to find the value you're looking for, and once it does, the value is returned). By adding an index to a column, the database keeps metadata about the structure of the column so that it can find the value you're looking for with many fewer data reads.

So, if adding indexes to a column can speed up query performance, how do we know if we need to add an index to a column? Get the database to explain it to you.

Explaining Postgres Performance

Postgres has a built-in `EXPLAIN` function that can be invoked in a query that returns the plan and execution time of the query.

For example, let's say we have a query that is taking longer than we would like:

```
SELECT * FROM users WHERE id = 1  LIMIT 1;
```

We can get more information by adding the `EXPLAIN` keyword before the query like this:

```
EXPLAIN SELECT * FROM users WHERE id = 1  LIMIT 1;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.00..8.27 rows=1 width=1936)
   ->  Seq Scan on users  (cost=0.00..9.68 rows=1 width=1936)
```

Here you can see that Postgres chose to sequentially scan the users table (i.e., it looks at each of the rows in the table until it finds the one you're looking for).

Sequential scans are expensive for large datasets and should be avoided. Frameworks such as Rails have the ability to "autoexplain" any queries that run over a given threshold while in development.

If you consistently see sequential scans over tables with many rows, consider adding indexes to speed up performance. When you do, Postgres will tell you when it is scanning using an index:

```
->  Index Scan using users_pkey on users  (cost=0.00..8.27 rows=1 width=1936)
```

So, when your data store starts getting slow, use logs and `EXPLAIN` to isolate and fix the problem. Once you've tuned your database, your queries may still have some calls to your database that will continue to be slow. Because the quickest database call is the one you never have to make, you can cache the results of long-running queries into a distributed cache store.

Caching Expensive Queries

Heroku offers many different add-ons that provide quick and efficient key/value stores that can be used to cache queries. Memcache and Memcachier are two add-ons that both provide access to a volatile cache store memcache. It is common for developers to use memcache to store "marshaled" data. (Marshaled data is persisted in a way that allows for simple transmission between systems: e.g., as a JSON or YAML serialization.) This provides developers with the ability to store not only primitive values like integers and strings, but also complex custom data objects such as users, products, and any other class imaginable (as long as your language provides the ability to marshal and then dump those data structures).

There are other key/value data stores known for their performance, such as Redis. We can't compare the costs and benefits of all of them here. The important thing is to remember that for calculations that are system intensive, caching the results can dramatically speed up your results.

Back that Task Up

Not everything your application does needs to be within the request/response cycle of an HTTP request. If a new user signs up to your application, she doesn't need to sit around and wait for a few milliseconds while your SMTP server responds because you're sending her a welcome email. Your visitor probably doesn't care if that email gets sent out immediately, so why are we making her wait for it before rendering the next page? Instead, kick this action off to a "background task" that can be handled by a separate "worker" dyno.

On Heroku, you can start a separate worker dyno by adding it to your job Procfile. If you're using Resque with Ruby, your Procfile might have something in it like this:

```
worker: bundle exec rake resque:work VERBOSE=1 QUEUE=*
```

Here you're telling Heroku to run the task `resque:work` in a set of dynos classified as `worker` and look at all of the queues `*`. There are a number of these worker libraries in

any number of different languages. The general concepts are the same. You store a task —such as sending out email, performing an expensive calculation, or making an API request—in some standard format in a shared data store such as Redis or PostgreSQL. The worker dyno reads the task data in from the data store and performs the necessary action. For some tasks, like email, the worker just needs to send an outbound communication, but for others it might need to write a value back to a database to save a calculation.

This will speed up individual requests, but it also improves your throughput, since each request now requires less computation. If you find that your background worker becomes slow or is not capable of catching up to all of the tasks given to it, you can scale it independently of your web dynos.

Another common component of applications is some kind of full-text search. Let's look at how we can move beyond `like` queries to go faster and return more relevant results.

Full-Text Search Apps

The dirty secret of most new web applications is that the search box prominently featured on the top of the page is in fact backed by a simple `like` database query:

```
select * from users where name like '%schneem%';
```

This strategy can work well for development and prototyping, but it scales horribly. The more entries you have in your database, the worse speed and relevancy will be.

The `like` query performs a sequential scan over your desired text. It also doesn't help you rank the relevancy of your results. Modern full-text search is capable of also "stemming" words so that it knows that "cars" and "car" are talking about the same root item. They also get rid of common low-quality words like "an," "the," "and," and so on.

On Heroku, you have two possibilities if you want to use full-text search. You can use a full-text search add-on such as web solr, Bonsai Elasticsearch, or Sphinx; or you can use the built-in full-text search in the PostgreSQL database.

If your database has heavily read traffic already, adding on a full-text search will only increase the load; instead, you may want to move this functionality to a full-text search add-on. None of these add-ons will turn you into Google overnight, but many of the underlying technologies have had years of optimization. Using default configurations makes it possible to get pretty good performance, though optimizations can produce huge gains no matter where you're doing your full-text search. The downside to using a full-text search add-on is that it adds an extra piece of complexity, and your app has to sync the text you want to be searchable.

Whichever option you end up choosing, don't ship your "search" feature powered by a `like` query.

Performance Testing

We've talked a lot about backend optimizations and frontend performance boosts, background workers, and full-text search engines, but at the end of the day the only thing that matters is that your site responds faster and remains scalable. It is important to measure your application's performance so you're able to determine how effective your improvements have been and to detect any regressions. This is easier said than done, as speed can be heavily influenced by network effects and the load the app is under during testing.

If you're interested in getting real-world performance measurements, there are two ways to keep track of your app: use in-app performance analytics services or use external load and performance testing tools.

In-App Performance Analysis

Wouldn't it be great if we could get our SQL log data and other backend performance characteristics in near real time with easy-to-understand formatted graphs? That's exactly what services like New Relic aim to do. Many languages and frameworks have instrumentation that allow a client running on production machines to measure and report to these services.

With these services, you can see details such as time spent fetching results from Memcache, running SQL queries, performing garbage collection, and anything else that your framework is instrumented to measure. You may choose to collect these numbers all the time in production, or only while you're actively working on performance tuning.

External Performance Tools: Backend

Aside from measuring your application performance from within your app, you can analyze the frontend and backend performance characteristics using third-party tools.

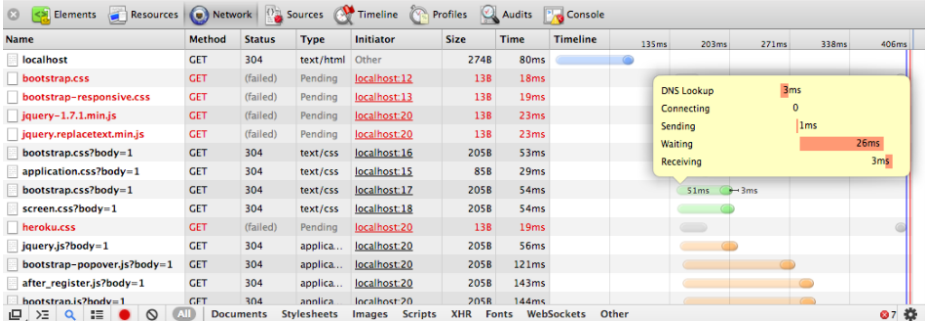
To test the throughput and response time of your app, you can use a service that generates traffic and hits your staging server. These services are best used for measuring backend speed because they don't take into account the time a browser takes to render a page. Heroku has several add-ons that can load and test your application, including BlitzIO and Loader.io.

Keep in mind that the performance of each run can vary, so you may want to perform these tests several times to get a good baseline. Once you have that data, you can use it to determine the effectiveness of speed-related changes. You can also use a load testing tool like this in combination with in-app analytics to determine which parts of your app are the first to slow down under increased load.

Load testing tools are a great way to get more visibility into your backend, but what about the page render time on the client side?

External Performance Tools: Frontend

Most browsers ship with some form of a request inspector that can be used to measure performance of page loads. For example, Chrome has a "Network" inspector. This Network tab will show you exactly how long an individual asset takes to download:



The Chrome inspector and similar tools for other browsers will show you where time is being spent, but they won't necessarily tell you what can be improved. One simple, yet useful, tool for this is YSlow, which will attempt to "rank" the frontend performance of your app and give you general advice on how to improve.

Being Fast

Applications aren't fast by accident; their developers take the time to understand and measure, and then improve areas found to be lacking. Performance is gained by careful work and can be lost with negligence. Setting aside time dedicated to measuring and improving speed is crucial. Consider taking a few hours at the end of the week with your team to look at application performance. Institutionalizing "Fast Fridays" can be a great way to shave off some of the performance debt that gets built up between rushed deadlines and shipped features.

If the performance of your application is crucial enough, you can consider instrumenting performance tests that benchmark and warn of dips in speed.

Finally, don't be afraid of application performance. Take slow, measured steps. Gather data, find the slowdowns, and then implement the fix. Applications don't become slow in a day, so expect to take some time to speed yours up. Performance is incremental. Enjoy the process of discovering more about your application, and be pleasantly surprised when your users start raving about how fast your app runs.

CHAPTER 4
Heroku Regions

These days, everything is global. Only 20 years ago a business may have limited its area of operation to only those places within a few hours' travel. Today, however, it is possible to trade globally and instantly. As for your applications, this means that you can interact with anyone, anywhere (your user base might consist of people in the United States, Europe, Australia, and Asia all at once).

So, what difference does this make to you? All of those users can access your application and use it—everything's fine, right?

Well, for starters, let's consider latency. Data travels down the pipe pretty fast, but there is a noticeable delay when working at very large distances. For instance, an application hosted on the eastern seaboard of the United States accessed from the United Kingdom has a latency of around 100 ms. The same application hosted in Europe, which is around 3,000 miles closer, only has a latency of around 20 ms.

When you are trying to wring every bit of performance from your application, this latency is not to be sniffed at—it's a sizable performance saving, and more or less for free. Just by changing where your application is physically can make a huge impact on your application performance and the happiness of your users.

So, what other reasons might there be for moving your application around the world? For instance, some nations or regions have laws in place to protect their citizens and limit what people can do with their data. For instance, an EU citizen is protected by laws that prevent companies from moving their personal data outside the jurisdiction of the European Union without that citizen's express permission. Here's an example: if a person living in the United Kingdom enters his data into an application whose data centers are in the United States, and he is not aware of this fact, the owner of the application in question would be in breach of these laws. Therefore, it is imperative in these instances that people consider where their application is hosted physically. The most common of this type of legislation is Safe Harbor, an agreement between the United States and the

European Union. Note that, at the time of writing, Heroku is not a registered Safe Harbor participant.

In the past, there has just been the concept of the cloud, but these days it is ever more important to consider the physical locality of where your application is hosted.

Multiregion or Single?

So, which do you want? Do you want to host your application in a single region such as the United States, or duplicate it in two such as the United States and Europe?

If you're looking to serve core customers in a different region from where your application resides, or there are legal data-hosting issues such as those described in this chapter, then you need to have a presence in that specific region.

In the past, this would have meant a significant headache. Not only would you have had to gather all the infrastructure required to host your application, as well as maintaining it going forward, but you'd then be required to do it all again in another region, but dealing with infrastructure partners who were, quite literally, foreign to you. This introduced even more complexity, as well as a whole load more challenges such as managing servers potentially thousands of miles away.

The Heroku Way

How does Heroku handle this problem? Simply put, Heroku is able to clone itself in new regions and provide the complete native stack in exactly the same way as in the original U.S.-East region. What's more, as all of Heroku is still controlled via a single API, there is almost nothing for you to do.

How to Deploy in a New Region

Let's look at the traditional process for creating an application using the Heroku command-line interface:

```
$ heroku create my_great_app
Creating my_great_app…done, stack is cedar.
http://my_great_app.herokuapp.com/ | git@heroku.com:my-great-app.git
Git remote heroku added
```

On its own, this command provisions your application on the Heroku platform in the default region of the United States and sits there ready for a deploy.

Now, let's take a look at how you might do this for deployment of an application to the EU region:

```
$ heroku create my_great_app --region eu
Creating my_great_app…done, stack is cedar.
```

```
http://my_great_app.herokuapp.com/ | git@heroku.com:my-great-app.git
Git remote heroku added
```

As you can see, this is almost identical. The simple addition of the region argument to your command changes where this application ends up, and where it serves from. What's more, this is identical in its functionality to its U.S. cousin.

At the time of writing, the available regions for deployment were limited to the United States and Europe.

So, how can you find out where an application is now that it is deployed? The region is shown in the application's info:

```
heroku info
=== calm-ocean-1234
Git URL:       git@heroku.com:calm-ocean-1234.git
Owner Email:   user@test.com
Region:        eu
Repo Size:     164M
...
```

Applications always deploy to the same region that they were created in. At the time of writing, there is no way to move applications between regions once created.

Forking an Application

So, what about existing applications? How can we get those into Europe (or vice versa)?

Well, this is why Heroku provides `fork`, a simple way of cloning an application from one location into another:

```
$  heroku fork -a neilmiddleton neilmiddleton-eu-region --region eu
Creating fork neilmiddleton-eu-region... done
Copying slug... done
Adding openredis:micro skipped (This app is in region eu, openredis:micro is only
 available in region us.)
Adding newrelic:standard... done
Copying config vars... done
Fork complete, view it at http://neilmiddleton-eu-region.herokuapp.com/
```

And that's it. This command copies your application across to a new application in the European Union, provisions all the add-ons you have, including databases, and copies your configuration to your new application.

Not all add-ons are available in Europe. Fork will inform you of this, as in the previous example.

Your new app won't be scaled past a single web dyno as per normal for new apps, and the fork won't have any of the domains attached to your old application. You're free to try out and test your new app at your leisure until you want to make it live yourself.

Constraints

Given that these processes are very similar regardless of where you are deploying from, what things do you need to be aware of when using this functionality? Luckily not much, and those that you do need to be aware of are simple to work around.

Latency

Latency is always something to consider when building global applications and something that needs to be factored into your application architecture when using regional deployments.

For instance, a common use of regions is to create a localized clone of your main production application in a region closer to a chunk of your user base. For example, you may have an application hosted in the United States that receives 40% of its traffic from Europe. In this case, it would make a lot of sense to create a localized version.

While this may initially sound simple, the application side is easier than dealing with data. Unless your application has a neat data structure that can be split, both regions are going to want access to the same dataset. This means that you need to make a decision about where to host your data.

Initially, your biggest problem is that hosting your data in one region will induce some latency for all of the other regions having to communicate back to your database in your primary region.

Read/write architecture

A way around these problems is to use a read/write architecture. In these systems, all of your database write queries go to one database, and all of your read queries go to another. This is commonly used in scaling applications where one master will be receiving writes, and the reads are distributed among multiple followers.

In terms of using this with Heroku regions, you can achieve much of the same thing; you can create a master database in the United States, and create a follower for that database in Europe:

```
$ heroku addons:add heroku-postgresql:ronin --follow --region eu-west-1 \
HEROKU_POSTGRESQL_CHARCOAL_URL
Adding heroku-postgresql:ronin to sushi... done, v71 ($200/mo)
Attached as HEROKU_POSTGRESQL_WHITE
Follower will become available for read-only queries when up-to-date
Use 'heroku pg:wait' to track status
```

If you've followed along, you'll have a primary database in the United States, and a follower in Europe, which—after a period of syncing latency—will contain the same data. Therefore, for EU customers, by sending all of our write queries to the U.S. database (which are generally infrequent for most applications) and our read queries to the EU follower, we are able to reduce the transatlantic latency for the majority of queries.

Note, though, that this is a very inexact science and depends heavily on the dynamics of your particular application. Some applications can separate their data entirely, thus removing all latency issues. Some applications will be very write heavy, thus increasing the problems with using a geographic follower.

In essence, the solution here is to experiment with different ways of structuring your application and determining what is acceptable for your own use case. Only then will you be able to appreciate where your application problems lie and how you might be able to solve them.

Add-Ons

One further consideration that you must take when carrying out regional deployments is that of add-on availability. As add-ons are provided by third parties, it is up to them to determine the locality of their services. Some add-on providers will provide their add-on locally to your application, some will choose to allow you to access them cross-region, while others will choose not to offer their add-on in specific regions at all. At the time of writing, most latency sensitive add-ons (e.g., databases, Redis, etc.) are available in both the US and EU regions, and nearly all non-latency-sensitive add-ons are available in both regions also.

The advice here is to check with the current state of add-on availability with Heroku at the time of making any decisions that this availability may affect.

CHAPTER 5

Heroku PostgreSQL

As Heroku grows, there is an increasing need to provide database services that match the ethos of the Heroku platform—simple provisioning and usage on demand at a low cost.

Thus, Heroku decided to develop the Heroku PostgreSQL service.

Why PostgreSQL?

Over the last few years, two open source database servers have become dominant on the Web. These servers are MySQL and PostgreSQL. In order to provide the best service available, Heroku decided to use PostgreSQL, and provide this service as the default choice for all database hosting.

In the early days of Heroku PostgreSQL, this was somewhat surprising, as MySQL was easily the most predominantly used database server at the time. However, over time, opinion has shifted more toward using PostgreSQL.

But why PostgreSQL?

The initial choice was made based on the fact that it was believed to be operationally more reliable than MySQL, which is very important when managing hundreds of thousands of databases on a daily basis. Over time, Heroku found PostgreSQL to get better and better as the development community around it added updated and new functionality.

Additionally, and a very key reason, PostgreSQL is an open system and always will be (unlike MySQL). This means that now and moving forward, as long as you are using PostgreSQL for your database server, you will not be subject to any vendor lock-in. Thus, as a user, you are able to take your data wherever you please.

There are a number of other technical reasons as to why PostgreSQL is the favored option, too: for instance, transactional data definition languages (DDLs), fast index creation with concurrent indexes, extensibility, and partial indexing and constraints.

Transactional DDL

If you've ever made a change to your database and have had something fail mid-way, either due to a constraint or some other means, you understand what pain can come of quickly untangling the resultant mess. Typically, changes to a schema are intended to be run holistically and if they fail, you want to fully roll back.

Some other databases, such as Oracle in recent versions and SQL Server, do support this. PostgreSQL, however, supports wrapping your DDL inside a transaction. This means if an error does occur, you can simply roll back and have the previous DDL statements rolled back with it, leaving your schema migrations as safe as your data, and your application in a consistent state.

Concurrent Indexes

When you create an index with most traditional databases, it holds a lock on the table while it creates the index. This means that the table is more or less unusable during that time. When you're starting out, this isn't a problem, but as your data grows and you add indexes to improve performance, it could mean downtime just to add an index (not ideal in a production environment).

Not surprisingly, PostgreSQL has a great means of adding an index without holding that lock. Simply invoking `CREATE INDEX CONCURRENTLY` instead of `CREATE INDEX` will create your index without holding the lock.

Of course, with many features, there are caveats. In the case of creating your index concurrently, it does take somewhere on the order of two to three times longer, and cannot be done within a transaction.

Extensibility

Do you need to go beyond standard PostgreSQL? There's a good chance that someone else has, and that there's already an extension for it.

Extensions take PostgreSQL further with things such as geospatial support, JSON data types, key/value stores, and connecting to external data sources (e.g., Oracle, MySQL, and Redis).

Partial Indexing

In a similar fashion to affecting only part of your data, you may care about an index on only a portion of your data, or you may care about placing a constraint only where a certain condition is true.

Take an example case of the white pages. Within the white pages, you only have one active address, but you've had multiple addresses over recent years.

You likely wouldn't care about the past addresses being indexed, but would want everyone's current address to be indexed. With partial indexes, this becomes simple and straightforward:

```
CREATE INDEX idx_address_current ON address (user_id) WHERE current IS True;
```

What Heroku Gives You

There are a number of reasons why you should use Heroku for hosting your PostgreSQL databases, some of which may not be immediately apparent. For instance, let's take a look at typical requirements of a database administrator (DBA):

- Set up and administer databases (perform backup, recovery, monitoring, and tuning as necessary).
- Devise a scalable solution with the highest performance and greatest redundancy.
- Deploy upgrades and patches.
- Implement and test disaster recovery.

In essence, a qualified DBA needs to be able to set up, manage, secure, back up, scale, architect, and optimize data structures for day-to-day use.

With PostgreSQL, as with the rest of the Heroku platform, most of this is done for you. In fact, Heroku will do everything aside from architect and optimize your database. There is no need to worry about patching for security or stability, and there is no need to worry about backups; everything is taken care of for you. All that is left for you to do is to worry about your data and how to store it efficiently.

Because Heroku takes on a lot of the responsibilities of a DBA, you are freed up to consider the parts that are important to your particular application and your particular use case. Meanwhile, Heroku is doing its part to keep your database healthy, functioning, and running well, 24 hours a day.

Development versus Production

At this point, it is worth mentioning that there are two classes of service provided by Heroku: development and production.

The Heroku PostgreSQL development plans are designed to offer the database features required for development and testing, without the production-grade operations, monitoring, and support found in paid production plans. Advanced features such as fork, follow, and automatic database backups are not available on the development plans (although manual backups are available).

Production plans are suitable for production-scale applications. Additionally, the production databases offer a number of advantages over shared, including direct access (via PSQL or any native PostgreSQL library), stored procedures, and PostgreSQL 9.1 support.

Choosing the Right Plan

Heroku PostgreSQL plans vary primarily by the size of their in-memory data cache. The quoted cache size constitutes the total amount of RAM given to the PostgreSQL service. While a small amount of RAM is used for managing each connection and other tasks, PostgreSQL will take advantage of almost all this RAM for its caching needs.

PostgreSQL constantly manages the cache of your data: rows you've written, indexes you've made, and metadata PostgreSQL keeps. When the data needed for a query is entirely in that cache, performance is very fast. Queries made from cached data are often 100 to 1,000 times faster than from the full dataset. Well-engineered, high performance web applications will have 99% or more of their queries served from cache.

Conversely, having to fall back to disk is at least an order of magnitude slower. Additionally, columns with large data types (e.g., large text columns) are stored out-of-line via The Oversized-Attribute Storage Technique (TOAST), and accessing large amounts of this data can be slow.

Unfortunately, there is no great way of knowing exactly how much cache you need for a given use case. This is not a very satisfactory answer, but it is the truth. The recommended way is to move between plans, watch your application performance, and see what works best for your application's access patterns.

Access patterns vary greatly from application to application. Many applications only access a small, recently changed portion of their overall data. PostgreSQL will automatically keep that portion in cache as time goes on, and as a result these applications can perform well on the lower cost plans.

Applications that frequently access all of their data don't have that luxury; these can experience dramatic increases in performance by ensuring that their datasets fit completely in memory. However, you will eventually reach the point where you have more data than the largest plan allows, and you will have to take more advanced approaches such as splitting your data across several databases.

Heroku Postgres's many plans are segmented in two broad tiers: starter and production. Each tier contains several individual plans. Although these two tiers share many features, there are several differences that will determine which plan is most appropriate for your use case.

Shared Features

The starter and production tier database plans all share the following features:

- Fully managed database service with automatic health checks
- Write-ahead log (WAL) off-premise storage every 60 seconds, ensuring minimal data loss
- Data clips for easy and secure sharing of data and queries
- SSL-protected PSQL/libpq ingress
- PostgreSQL extensions

Together, these features combine to provide a safe and resilient home base for your data.

Starter Features

The starter tier database plans are not intended for production-caliber applications or applications with high-uptime requirements. Therefore, if you are hosting an application that you consider to be critical, it is advised that you consider using the production tier database plans.

The starter tier, which includes the dev and basic plans, has the following limitations:

- Enforced row limits of 10,000 rows for dev and 10,000,000 for basic plans.
- Max of 20 connections.
- The lack of an in-memory cache limits the performance capabilities since the data can't be accessed on low-latency storage.
- Fork and follow, used to create replica databases and master-slave setups, are not supported.
- Expected uptime of 99.5% each month.

In addition to these feature and resource limitations, starter tier database plans do not automatically record daily data snapshots. You will need to use the PGBackups add-on to manually configure this level of data protection and retention.

Production Features

As the name implies, the production tier of Heroku PostgreSQL is intended for production applications and includes the following feature additions to the starter tier:

- No row limitations
- Increasing amounts of in-memory cache
- Fork and follow support
- Max of 500 connections
- Expected uptime of 99.95% each month

Management of production tier database plans is also much more robust, including:

- Automatic daily snapshots with one-month retention
- Priority service restoration on disruptions

Getting Started

So how do we provision and use a Heroku PostgreSQL instance? Fortunately, this is a very simple task.

A prerequisite of following this walkthrough is to have a Heroku account, verified with a credit card. To sign up, if you haven't already, visit *http://postgres.heroku.com/signup*.

To provision a database, follow these steps:

1. Go to your database dashboard, the page that loads immediately after logging in.
2. Click the + in the header.
3. Select the Dev Plan (which is free) when you are presented with a list of the options available.
4. Click Create Database. You should now see a message that the database is being created. A few seconds later, this page will change to show you the name of your new database (e.g., "protected-wave-27").
5. Click the database name.

Your PostgreSQL database is now provisioned and ready to use. From the current screen, you can see information such as the host, database name, and credentials needed to connect.

By clicking the cog at the upper-right corner of the Connection Settings panel, you are able to select one of the commonly used methods of connecting to your database, and get a pregenerated string that you can use in your application. For instance, if you were to select ActiveRecord (the Ruby on Rails ORM), you would see something similar to this:

```
adapter:        postgresql
encoding:       unicode
pool:   5
database:       db3bjv2gmckc5c
username:       fKpF7NNtm4kfZd
password:       7ARGxieXAnZEyyLMm9h
host:   ec2-23-21-85-231.compute-1.amazonaws.com
port:   5432
```

You will also be able to see other information, such as the database type and some basic statistics regarding size, version, and availability. From here, you are also able to see your current data clips and snapshots. More on that later.

Importing and Exporting Data

Once you have your database, how do you get your preexising data in? How would we get our data out of Heroku PostgreSQL for further querying offline? In order to make this process as efficient as possible, Heroku has made sure that there is no lock-in, and made sure that you can access your data in the normal PostgreSQL ways.

Importing Data

In many cases, you will now want to import some data into your database for use by your application. This is done via the use of a dump file from another PostgreSQL database:

```
$ pg_dump -Fc --no-acl --no-owner my_old_postgresql > data.dump
```

Alternatively, a dump can be created via the use of PGAdmin, the GUI tool for PostgreSQL administration. To do so, select a database from the Object Browser and click Tools → Backup. Set the filename to *data.dump*, use the COMPRESS format, and (under Dump Options #1) choose not to save Privilege or Tablespace.

The primary option for importing this data into your database is to use `pg_restore`.

Using pg_restore locally

The PostgreSQL `pg_restore` utility can restore your *data.dump* file to your database locally.

To install the `pg_restore` utility, you must install PostgreSQL locally. More information on doing this can be found here.

With a `DATABASE_URL` of `postgres://<USER>:<PASS>@<HOST>:<PORT>/<DBNAME>` the `pg_restore` command would be:

```
$ PGPASSWORD=<PASS> pg_restore --verbose --clean --no-acl --no-owner -h <HOST> \
   -U <USER> -d <DBNAME> -p <PORT> ~/data.dump
```

Be sure to replace the appropriate elements with those given in your connection settings.

For now, though, let's import a sample 200 MB employee database that contains the following:

- Employees
- Departments
- Salaries (many-to-one with employees)
- Titles (many-to-one with employees)
- Department employees (one-to-one with departments and employees)
- Department managers (one-to-one with departments and employees)

First, download the dump file:

```
$ curl -o employees.dump https://heroku-data.s3.amazonaws.com/employees.dump
```

Then, use `pg_restore` to restore directly to your Heroku PostgreSQL database. Remember, the correct `pg_restore` syntax can be automatically generated from your database detail page:

```
$ PGPASSWORD=<PASS> pg_restore --verbose --clean --no-acl --no-owner -h <HOST \
   -U <USER> -d <DBNAME> -p <PORT> employees.dump
```

Once complete, you should be able to connect to your database, and query the database as if it were local.

Exporting Data

So, now that you've got some data in your database, how do you get it out again? You could query it and write it all down on some slips of paper, but this will take a fair while, so it's probably best to use one of the many more efficient methods.

Snapshots

Let's say you need to do a full database export. You can download a database snapshot, which is essentially identical to the dump file that you created when importing. To create a snapshot, log in to your Heroku PostgreSQL account and go to your database list. Select the database you're interested in and click the + in the Snapshots section. A few seconds later, the snapshot will have been created and can be downloaded.

Once downloaded, restore the dump to your local database as normal.

CSV Exports

But what if you don't want to download the whole database? Heroku will let you export the results of a single query as a comma-separated values (CSV) file via a SQL command (as built into PostgreSQL itself). In order to do this, connect to your PostgreSQL database (as described earlier in the chapter) and open up a new query window. Once done, issuing a query such as the following will create a new file locally containing the query data:

```
COPY (SELECT * FROM products) TO dump.csv DELIMITER '\t'
```

where:

```
SELECT * FROM products
```

is the query that you wish to download.

PGBackups

A very common requirement is backups. Although Heroku strives to ensure that there is no data loss in the event of some sort of digital catastrophe, it's always a good idea to make sure you keep backups for your own peace of mind.

In order to use PGBackups, you must first provision the add-on. Note that you will have no access to data backups without this add-on:

```
$ heroku addons:add pgbackups
-----> Adding pgbackups to my_app done, v18 (free)
```

In order to create a backup manually (PGBackups also supports automated backups), you can simply request it from the command line:

```
$ heroku pgbackups:capture

HEROKU_POSTGRESQL_BLACK (DATABASE_URL)  ----backup--->  b251

Capturing... done
Storing... done
```

Should you require, you can then download your most recent backup:

```
$ heroku pgbackups:url
"http://s3.amazonaws.com/hkpgbackups/app1234567@heroku.com/…
```

If, however, you require access to anything other than the most recent backup, you can review your archive:

```
$ heroku pgbackups
ID   | Backup Time         | Size  | Database
-----+---------------------+-------+-----------------------
a226 | 2013/02/22 20:02.19 | 5.3KB | DATABASE_URL
a227 | 2013/02/23 20:02.19 | 5.3KB | DATABASE_URL
b251 | 2013/02/24 16:08.02 | 5.3KB | HEROKU_POSTGRESQL_BLACK
b252 | 2013/02/24 16:08.53 | 5.3KB | HEROKU_POSTGRESQL_PINK
```

Data Clips

One common need of a database administrator is to share data in her database with other people. Generally speaking, as these people are rarely technical, sharing dump files with them might not be the best approach. What's more, it is generally a common need that this data is provided on a periodic basis. For instance, you may need to deliver an export of year-to-date sales by month to your manager each and every month.

Sharing information on the Internet is done by sharing URLs. URLs identify locations, books, videos, and even source code. Until now, there hasn't been a convenient way to share data inside a database. That's why Heroku introduced data clips. They are a fast and easy way to unlock the data in your database in the form of a secure URL.

Data clips allow the results of SQL queries on a Heroku PostgreSQL database to be easily shared. Simply create a query against a database within your databases list, and then share the resulting URL with coworkers, colleagues, or the world.

Data clips can be shared through email, Twitter, irc, or any other medium, because they are just URLs. The recipients of a data clip are able to view the data in their browser or download it in JSON, CSV, XML, or Microsoft Excel formats.

As data changes rapidly in databases, so can data clips. They can either be locked to a point in time or set to refresh with live data. When locked to a point in time, data clips are guaranteed to show an unchanging snapshot of data, even if they are viewed hours, days, or years after the clip was created. Alternatively, when data clips are set to "now," they provide a live view into the database in its current form.

Followers

So, now that we've talked about some of the data-in and data-out tasks that we commonly have to do day to day, let's talk more about the features that relate more to running Heroku PostgreSQL databases in an operational environment.

Although your database is hosted by Heroku and administered by some of the best DBAs in the business, there is still risk. Your database is running on one machine for instance, and should this go pop, then you're looking at some downtime until the database is moved to another location. This is why Heroku developed followers.

Followers are essentially the same as replicated databases. This means that you have a single master database that receives read and write queries and a follower, which is another identical database that mirrors the master database from a data point of view.

There are a few reasons to do this. First, as upgrading (or downgrading) your database requires replacement of the database itself, creating a follower first in the new plan allows you to simply switch across and then decommission the old database with very little downtime.

Another reason is to allow you to create a backup database that you can have in case of a failure of the first. You could use followers to create a sharding system where write queries are directed at the master, and read queries are directed at a follower to alleviate load on a single database.

Note, however, that followers cannot be instantly created, and that they take time to populate. Therefore, once you've created a follower, you will need to interrogate its completeness via the Heroku command-line interface:

```
$ heroku pg:info --app sushi
=== HEROKU_POSTGRESQL_RED
Plan         Ronin
Status       available
Data Size    82.8 GB
Tables       13
PG Version   9.0.5
Created      2011-06-15 09:58 PDT
=== HEROKU_POSTGRESQL_WHITE
Plan         Ronin
Status       following
Data Size    82.8 GB
Tables       13
PG Version   9.0.6
Created      2011-11-15 09:54 PDT
Following    HEROKU_POSTGRESQL_RED (DATABASE_URL)
Behind By    125 commits
```

Fast Database Changeovers

By now, you should see that it is possible to swap out your database for a new, more powerful one with very little downtime.

This changeover uses followers to minimize the downtime in migrating between databases. At a high level, a follower is created in order to move your data from one database to another. Once it has received the majority of the data and is closely following your

main database, you will prevent new data from being written (usually by enabling maintenance mode on your app). The follower will then fully catch up to the main database and be promoted to be the primary database for the application.

One thing that makes this entire process much simpler is to temporarily set your application as read-only. This won't suit some applications, but making sure that your data isn't changing will ease the process significantly.

In order to carry out a database changeover, follow this simple process:

1. Create a follower for your database.
2. Wait for the follower to catch up using the technique we've explained.
3. As it's important that no transactions are changing data, enable maintenance mode on your application:

   ```
   $ heroku maintenance:on --app sushi
   ```

   And scale every dyno down (in this instance, the `workers` and `fast_workers` dynos):

   ```
   $ heroku ps:scale workers=0 fast_workers=0 --app sushi
   ```

4. Confirm that your follower is zero commits behind your master and unfollow your main database:

   ```
   $ heroku pg:unfollow HEROKU_POSTGRESQL_WHITE --app sushi
   ```

5. Promote your follower to the main database:

   ```
   $ heroku pg:promote HEROKU_POSTGRESQL_WHITE --app sushi
   -----> Promoting HEROKU_POSTGRESQL_WHITE to DATABASE_URL... done
   ```

6. And return your application to normal:

   ```
   $ heroku ps:scale workers=X fast_workers=Y --app sushi
   $ heroku maintenance:off --app sushi
   ```

7. At this point, your original main database is now unused and can be exported, destroyed, or simply left as is. Note, though, that you will still be charged.

Forking

One last feature that is worth mentioning is *forking*. There is a common need to debug issues that occur in a live production environment—typically issues whereby a user has somehow managed to get himself into a certain state. In these situations, the problems are usually data related, so the developer tasked to debug the issue needs to have access

to this data. You could download this data and re-create it locally, but if your database is large this could be very inefficient.

Another scenario could be that you have a deployment coming up and you'd like to test some database changes against the production data before going live with the changes. Again, downloading a database that could be gigabytes in size would be a bad idea.

Therefore, Heroku has developed fork, which is essentially the practice of taking an existing Heroku PostgreSQL database and making a direct copy of it and the data associated with it at that particular time. This copy does not follow your main database or change in any other way, so it's a good way of cloning a database for interrogation or testing against.

As these forked databases are just like any other database, you can connect a staging version of your application to it and run your code as normal, all without affecting your production application.

To fork a database:

```
$ heroku addons:add heroku-postgresql:ronin --fork HEROKU_POSTGRESQL_RED
-----> Adding heroku-postgresql:ronin to sushi... done, v72 ($200/mo)
       Attached as HEROKU_POSTGRESQL_ORANGE
-----> Database will become available after it completes forking
   Use 'heroku pg:wait' to track status
```

Preparing a database fork can take anywhere from several minutes to several hours, depending on the size of your dataset.

Other Features

Aside from those already described, the Heroku team is constantly adding new features to their PostgreSQL offering on a weekly basis. This partly comes from the benefit of managing one of the largest fleets of PostgreSQL databases on the Web. The team, by virtue of the sheer number of databases under its control, is able to gather a vast amount of usage data, and a comprehensive list of pain points that other users are suffering. By using this data and contributing back to the open source PostgreSQL project, the platform can constantly improve.

Let's look at some of the key added features that PostgreSQL offers.

Extension Support

Databases are the well-known solution for storing data for an application. However, they sometimes lack functionality required by application developers, such as data encryption or cross-database reporting. As a result, developers are forced to write the needed functionality at their application layer. PostgreSQL 9.1, which already has an extensive collection of data types and functions, took the first step toward mitigating

this by creating an extension system that allows the database's functionality to be expanded.

Extensions allow related pieces of functionality, such as datatypes and functions, to be bundled together and installed in a database with a single command.

Heroku began supporting extensions in March 2012 with the release of *hstore*, the schemaless datatype for SQL. Users have taken advantage of hstore to increase their development agility by avoiding the need to predefine their schemas.

To install an extension, use the `CREATE EXTENSION` command in *psql*:

```
$ heroku pg:psql --app sushi
psql (9.1.4)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)

=> CREATE EXTENSION citext;
CREATE EXTENSION
```

A list of available extensions can be found at the Heroku DevCenter.

Improved Visibility

Visibility into your data has long been a problem for many application developers. In the current version of PostgreSQL (9.2 at the time of writing), all queries are normalized and data about them is recorded. This allows you to gain insight such as:

- How often is a query run?
- How much time is spent running the query?
- How much data is returned?

Each of these key pieces of data is critical when it comes to effectively optimizing your database's performance.

The old way of poring through logs is no longer needed to gain this insight. Now your database contains what it needs to help you improve performance within a PostgreSQL database.

Ensuring such functionality is committed back to the PostgreSQL core is very important, as it prevents lock-in and creates a better ecosystem for the community as a whole.

Let's take a look at how we can begin using some of this. First turn on statement tracking with `CREATE EXTENSION pg_stat_statements;`. Then run the next query and you'll receive all of your top run queries:

```
SELECT
    count(*),
    query
FROM
  pg_stat_statements
GROUP BY 2
ORDER BY 1 DESC
LIMIT 10;
```

JSON Support

Developers are always looking for more extensibility and power when working with and storing their data.

With PostgreSQL 9.2, there's even more robust support for NoSQL within your SQL database in the form of JSON. By using the JSON datatype, your JSON is validated as proper JSON before it's allowed to be committed.

Beyond the datatype itself, there are several new functions available. These are `re cord_to_json`, `row_to_json`, and `array_to_json`. Using these functions we can turn a row/record, or even an array of values, immediately into JSON to be used within an application or returned via an API:

```
$ heroku pg:psql
=> SELECT row_to_json(row('foo','bar', 1, 2));
     row_to_json

 {"f1":"foo","f2":"bar", "f3": 1, "f4": 2}
(1 row)
```

Range Type Support

The range datatype is another example of powerful data flexibility. It is a single column consisting of a to-and-from value. Your range can exist as a range of timestamps, can be alpha-numeric or numeric, and can even have constraints placed on it to enforce common range conditions.

For example, this schema ensures that in creating a class schedule we can't have two classes at the same time:

```
CREATE TABLE schedule (class int, during tsrange);
ALTER TABLE schedule ADD EXCLUDE USING gist (during WITH &&);
```

Then attempting to add data we would receive an error:

```
INSERT INTO schedule VALUES (3, '[2012-09-24 13:00, 2012-09-24 13:50)');
INSERT INTO schedule VALUES
(1108, '[2012-09-24 13:30, 2012-09-24 14:00)');
ERROR:  conflicting key value violates exclusion
        constraint "schedule_during_excl"
```

Other data types appear all the time, so it's well worth keeping an eye on Heroku's DevCenter and blog for announcements of further types.

CHAPTER 6
Deployment

Although deploying to Heroku is a simple process, people commonly forget to consider some things when deploying applications, not only to Heroku, but anywhere else as well. Some of these items are not necessarily technology driven, and form the basis of a good checklist that you can use to ensure that your deployments are of a much more organized nature. We'll take a look at some of these issues now. When deploying, you want to ensure that your deployments are safe and timely, but you should also have a Get Out of Jail Free card handy should it not go as planned.

Timeliness

Let's start with one of the most basic considerations, which can alleviate a lot of pain surrounding deployments, particularly to production environments: *when* to deploy.

It is common for client requirements to dictate that deployments happen at the end of the day or week, and that's something that a lot of people will happily do—however, this is probably the worst possible time to deploy. Deploying at the end of the day or week leaves you open to all sorts of issues arising, and having nobody around to help resolve them. Therefore, always plan on deploying when you've got a large window of availability in order to resolve any problems that might crop up. Having your application go down while you're happily driving home for the weekend does not make for a good start to your time off.

Backing Out

Another common thing that is overlooked in the sometimes frantic rush to deploy is being able to back out a deployment should things go wrong. Typically, a deploy will not only change code, but also data and potentially cache contents. Therefore, it is always a good idea to have a plan of how you might recover things should the deploy not go well.

Heroku makes release rollback simple via the releases commands in the command-line interface:

```
$ heroku rollback
```

But you should also consider how to recover other parts of your application. For instance, do you have a backup of your database in its current state that you can use should your new data migrations fail? Heroku Postgres Fork is superb for this. Do you have a way of clearing caches and letting them rebuild from the current state if your code isn't using the old caches properly? A few minutes forethought on backing out your releases can often be worth its weight in gold.

Testing

A third item that is often missed is testing your deployments. Developers are often great at building immense code coverage with unit testing, but commonly fail to think about testing that deployments work as they should and that no unforeseen problems occur. Finding these issues in a production environment does not often leave you with a smile on your face. So, how can you mitigate this risk?

Heroku makes this very easy by letting you create separate environments and forking applications. Futhermore, it's relatively simple to create an environment exactly like the one you have in production, but with a different URL. Forking allows you to duplicate your production data, and using Git for source control lets you deploy exactly the same code to your new environment before testing your new deployment. For more information on forking, refer back to "Forking an Application" on page 35.

Setting up a staging environment may take a few minutes, but it can pay dividends in the long run if you uncover issues that you hadn't thought of.

How to Deploy, and Deploy Well

So what do we need to consider when deploying an application? We could just push our code to Heroku and hope for the best, but things go wrong. Someone may have accidentally pushed code not ready for production, or we may have some potentially data destroying bug in our migrations. How can we be sure that we have a way out if things go bad?

Backups

It is good practice to back up before you make any changes, either locally in development or on Heroku in your production environment. Losing data through corruption or overwrite is every business's worst nightmare, so it is worth investing a few seconds to ensure that you are able to recover your data and code to a previous version should

things go awry. For more information on how to back up data with the Heroku Postgres service, see "PGBackups" on page 47.

Heroku Releases

Whenever you deploy code, change a config var, or add or remove an add-on resource, Heroku creates a new release and restarts your app. You can list the history of releases, and use rollbacks to revert to prior releases for backing out of bad deploys or config changes.

Release creation

Releases are named in the format `vNN`, where **`NN`** is an incrementing sequence number for each release. Releases are created whenever you deploy code. In this example, `v10` is the release created by the deploy:

```
$ git push heroku master
...
-----> Compiled slug size is 8.3MB
-----> Launching... done, v10
    http://severe-mountain-793.herokuapp.com deployed to Heroku
```

To see the releases for an application:

```
$ heroku releases
Rel   Change                        By                  When
----  ----------------------        ----------          ----------
v52   Config add AWS_S3_KEY         jim@example.com     5 minutes ago
v51   Deploy de63889                stephan@example.com 7 minutes ago
v50   Deploy 7c35f77                stephan@example.com 3 hours ago
v49   Rollback to v46               joe@example.com     2012-09-12 15:32:17
```

And to get detailed info on a release:

```
$ heroku releases:info v24
=== Release v24
Change:      Deploy 575bfa8
By:          jim@example.com
When:        6 hours ago
Addons:      deployhooks:email, releases:advanced
Config:      MY_CONFIG_VAR => 42
             RACK_ENV      => production
```

Rolling back

Rolling back will create a new release, which is a copy of the state of code and config vars contained in the targeted release. The state of the database or external state held in add-ons (e.g., the contents of memcache) will not be affected and are up to you to reconcile with the rollback.

Running on rolled-back code is meant as a temporary fix to a bad deployment. If you are on rolled-back code and your slug is recompiled (for any reason other than a new

deployment) your app will be moved back to running on the most recent release. Subscribing to a new add-on or adding/removing a config var will result in your slug being recompiled.

Performance Testing

Now we have our code, and we know it works locally for us and our test suite passes with no problems, but how do we know that our code will function under load? How do we know that, once we have a thousand users hitting our application, we're able to perform in a way that is acceptable and doesn't end up losing us business at the most critical times? The only way to be sure is performance testing.

Performance testing is a vast subject that is outside of the scope of this book, but you can take some very simple steps to check the basics. Load testing is the simplest form of performance testing. By exposing your application to an expected amount of load you can witness how your application will perform and where the potential bottlenecks are. Another common method is stress testing. Unlike load testing, which is testing your application under a given load, stress testing pushes your application as far as it can go before breaking. The aim of the stress test is to take your application to the point where it cannot go any further, thus exposing you to its upper limits. By a combination of load and stress testing, you are able to determine how your application will perform at peak times; it also gives you an idea of how much extra capacity you have in your pocket should the time arise.

There are a wide variety of sources for more information on performance testing. A good place to start is *The Art of Application Performance Testing* by Ian Molyneaux (O'Reilly, 2009).

Trimming the Fat (Slug Management)

One aspect that can improve the management of your application significantly is the time that it takes to deploy. When deployed to Heroku, your application resides in a slug archive stored inside a massive file store. When scaling, this archive is then copied from this file store and deployed on a fresh dyno. Once on this dyno, it's spun up and starts serving requests. For day-to-day deployment and scaling, the time this process takes can have a significant effect on managing your application. For instance, imagine your application's slug was extremely large; this slug would take time to copy across the network, time to unzip, and time to spin up. This means that between hitting the *scale up* button, and the new dyno being available, you could be looking at a significant amount of time.

A simple way of improving this is via the *.slugignore* file. In essence, *.slugignore* is the same as a *.gitignore* file (Git) or an `svn:ignore` property (Subversion). This file tells Heroku which files it can ignore from your application's source repository, and

therefore, which not to include inside your applications slug. One very common use case is that of the test suite. Test suites can sometimes be as large as your application itself, and can also contain large files such as test images or documents to test imports. By adding your suite to the *.slugignore*, this won't comprise part of your slug once deployed to Heroku.

Luckily, *.slugignore* uses exactly the same format as *.gitignore* (barring the ! negation operator) so the file format should already be familiar to most:

```
*.psd
*.pdf
test
spec
```

When deploying to Heroku, your slug size will be displayed in the output, so it's good to keep an eye on this and ensure that you're staying within acceptable limits. Generally speaking, any slug under 15 MB is small and nimble; 50 MB is average; and 90 MB or above is weighty.

Source Code Management

While source code management is integral to developing and deploying with Heroku, it is far too vast a topic to cover in this book, and there are several books out there that cover it extremely well. However, some aspects of managing your source code are very helpful when working with Heroku and are wise to consider, regardless of what other best practices you may already be following.

Bear in mind that as Git is the de facto source control system on the Heroku platform, this is the system we're describing here. When we're talking about branching, we realize that within Git this is an incredibly inexpensive process (unlike with some other software configuration managers out there).

Branching for environments

Probably the most significant thing that you can do to make your life simpler is to create branches within your source control for each environment that you have running on Heroku. For instance, you may have a production environment, a couple of staging environments, and maybe even the odd QA or acceptance environment. By creating a branch in your source control for each of these environments and religiously deploying from the appropriate branch to the correct environment, you are able, at a glance, to see what code is in which environment. Additionally, you're able to easily move code from one environment to another. Have a QA environment ready to go to staging? Merge the `QA` branch into the `Staging` branch, and deploy that branch to the staging environment. Need to test a bug in your production environment in a new `QA` branch? Create a new branch from your `Production` branch, deploy it to a new environment, and test away!

Feature branching

In addition to branching for each environment you own, it's also wise to branch for each significant feature that you're building in your application. Features are not always deployed in the order they are built, so if you're developing features one after the other inside your staging branch, you then lose control over how these features make it out into production or similar. By maintaining these features in separate branches, you are able to merge them individually into your environment branches; this is indispensable, especially when your stakeholders are indecisive or external factors have an impact on feature releases.

There are exhaustive references written on how to effectively use branching in source control with a team. We don't have the time here, so instead we recommend researching Git Flow for dealing with feature branches, as well as its slightly modified cousin GitHub Flow. Understanding branches is critical to effectively working in software.

Multienvironment Deployment

At this point, it should be clear that you don't need to limit yourself to just one running instance of your application on Heroku. As you currently get 750 hours a month of free dyno time per application per month, you are able to spin up an instance of an application and run it on a single dyno for free 24 hours a day. What's more, an application still exists even without any running dynos, so scaling everything down when it's not needed means that you're not using any hours at all. This means that the days of test and staging environments being limited in availability are long gone, and these environments now should be treated as freely available and disposable.

How can you leverage this ability to have your application running in multiple places at once? For starters, let's revisit the basics of deployment and learn a little bit about Git remotes.

Consider the most basic command for deploying to Heroku:

```
$ git push heroku master
```

This default git command is telling Git to do something very specific. In a simple sense, it's asking to push the `master` branch to the `heroku` remote, a remote simply being a foreign Git repository somewhere out there on the wire. You can see the definition of this remote by looking at your git configuration:

```
$ git remote -v
origin  git@github.com:neilmiddleton/my-sample-app.git (fetch)
origin  git@github.com:neilmiddleton/my-sample-app.git (push)
heroku  git@heroku.com:morning-sunshine-42.git (fetch)
heroku  git@heroku.com:morning-sunshine-42.git (push)
```

Here you are seeing two remotes. First, `origin`, which is a repository on GitHub, and second, a remote called `heroku`, which is automatically created when the application is created with `heroku create`.

The name of this `heroku` remote, however, is arbitrary; it's just a string identifier for this remote. If you wanted, you could rename this remote to whatever you chose; for instance, `production` would be equally as useful.

By adding new remotes to your application, you are able to push your code to other places as easily as our first deploy. Let's say you want to create a new staging environment for this application. First, you can create another new application and give it a specific name on Heroku and ask for a specific remote name:

```
$ heroku create my-sample-app-staging -r staging
```

This creates the application `my-sample-app-staging` and adds it as a remote to the current application. Once this has completed, you can then do a simple deploy:

```
$ git push staging master
```

and the magic will happen.

With this in mind, you can go as far as you want: you can create remotes for every environment you might ever need, and push to each of those remotes as much as you want. However, this only works for deploys. What about all the other Heroku commands?

Well, each and every command you can issue to Heroku via the command line takes an extra argument of the application in question:

```
$ heroku logs --app my-sample-app
$ heroku logs --app my-sample-app-staging
```

and so on.

By using Git remotes and targeting your commands to the individual instances of your application, you create, in theory, as many copies of your applications as you desire and manage them all separately.

Consistency

A common pitfall for people using this multiinstance approach, and one that we authors have fallen into many times, is not maintaining consistency across all the instances as much as possible. For example, if you add an add-on to one application, ensure that it is also on all the other applications as they require them.

Failure to do this can lead you down a path of finding difficult-to-diagnose bugs, or environments that simply don't function in the same way, which can very easily become frustrating.

Teams

Something that you'll find yourself doing at one time or another when working with Heroku is collaborating with other people on an application in some way. There are a number of things to consider when doing this, and luckily the vast amount are common sense.

For instance, ensure you have a sound deployment workflow whereby code cannot make it from development to production without being checked at least by automated testing, or another pair of eyes. Communication can help here for ensuring that the wrong things aren't deployed at the wrong times, but workflow can pretty much guarantee that bad things can't happen unless you happen to have failed several steps of your workflow.

Branching strategies such as those described in this chapter will go a long way here, as will staging environments combined with some kind of stakeholder sign-off on the changes that are moving their way through the system. Ultimately, though, Heroku enables anyone in a team to be as empowered as any other, so it's wise to consider what internal rules you might put into place before inviting the hordes to collaborate on your production application.

DNS

One of the hoops that every developer needs to jump through when putting an application into production is setting up his DNS so that *www.yourapp.com* resolves to his Heroku application and serves pages as it should.

Normally this is a very straightforward task when setting up on a simple VPS. Your VPS has an IP address, and you create some A records on your domain to point your domain at that server.

However, this gets a little more complicated when you need to scale. Let's say you have to add a second server to your stack—you're now looking at some sort of round-robin DNS, or implementing a load balancer and moving all of your DNS traffic through that new device.

Now let's say you want to scale to the level of something like Heroku (it's no simple task). Let's consider how the Heroku DNS works. At the front of the stack is a massive array of load balancers that are receiving traffic from the outside world and directing it at the routers. To the outside world, these load balancers have IP addresses, and can be pointed to with A records.

However, this is a very bad idea. There are literally millions of applications being routed at a rate of tens of thousands of requests per second through these load balancers, so Heroku constantly has to reconfigure this layer to ensure that the traffic being received is being served quickly and is stable. What's more, when some script kiddie out there

decides to have some fun, he might launch a denial of service attack against these load balancers. At the point the Heroku operations team will react and make the changes that need to be made to keep things alive and stable.

Configuration

So, how do we configure our DNS so that we don't have to worry about all of this? Well, every application has a **.herokuapp.com* domain that is managed within Heroku's own DNS infrastructure centrally. Heroku ensures that these applications always have a healthy point to which to send the user and DNSs are updated as needed. This means that these **.herokuapp.com* domains are an ideal place for you to send your users.

Therefore, when setting up your own DNS, it is always a good idea to use CNAMEs to alias your domain to the *herokuapp.com* domain. CNAME records are essentially a simple redirection—DNS-speak for saying *my config is the same as this other domain's, so use that*. This means that you also benefit from changes that Heroku is making but don't need to worry past the basic setup.

Apex domains

But what about root/apex domains (e.g., *yourapp.com*, not *www.yourapp.com*) that aren't allowed to use CNAMEs? Here you have a few choices: either you can use regular A records and point those at the few IPs that Heroku publishes in its DevCenter documentation, or you can try not to use apex domains and try and encourage your reluctant users with a subdomain prefix such as *www*.

A more foolproof way is to use some of the more modern hosted DNS offerings out there. Some companies such as DNSimple now offer their own custom types of records in your DNS that allow you to alias apex domains in the same way as a CNAME, and they handle this DNS magic within their own infrastructure. Setup is simple: sign up for one of these services, read the documentation, point your domain name servers at theirs and set up your domain as required. What's more, these services cost peanuts for the average user.

Setting up DNS in a way that's fully compatible with the Heroku platform is relatively simple, but it can be a bit daunting at first. There's plenty of documentation in the DevCenter on the topic, plus lots of documentation on the hosted DNS services such as DNSimple. The key thing to remember is that you're setting up your DNS to be infinitely more resilient than pointing it to a single IP, and also doing it in a way that will let you scale as far as you could ever need.

Let's go through a couple of common examples: one where you wish to host your website on the apex domain (*heroku.com*), and one on a subdomain (*www.heroku.com*). For both examples, we'll use the tools available to us from DNSimple:

Hosting on apex

In these instances, we need to use the `ALIAS` record so that we can bind our apex domain to our Heroku application:

```
my_app.com   ALIAS  my_app.herokuapp.com
```

For any subdomains, we can do several things. We can either redirect the *www* subdomain to the apex via the URL record:

```
www.my_app.com  URL  my_app.com
```

This will redirect the user to the apex, or we can CNAME it:

```
www.my_app.com  URL  my_app.herokuapp.com
```

The problem with CNAME here is that you will have issues with canonical URLs, as the site will be available on both the apex and *www* subdomains.

Hosting on a subdomain

Ideally, you want to be hosting your website on a subdomain such as *www*. This makes the setup much simpler:

```
www.my_app.com  URL  my_app.herokuapp.com
```

To deal with the apex, we just need to redirect to the appropriate subdomain, although this isn't strictly required:

```
my_app.com  URL  www.my_app.com
```

This doesn't have any issues with canonical URLs, and also means that search engines will index the subdomain, something that will help you in the future should you need to rearrange your application.

CHAPTER 7

When It Goes Wrong

Exceptions happen, even on exceptionally well-written applications. But when the unthinkable happens, how can you find your problems and fix them? Let's dig into debugging problems on deploy and during runtime.

Dev Center

Heroku maintains a wealth of documentation via articles about all sorts of topics related to the platform and the various tools that surround it. If you're having problems, this should be your first port of call as you should be able to find detailed information about the moving parts surrounding the issue you are experiencing.

The Heroku DevCenter can be found here.

Heroku Support

Remember, you're not on your own. Once you've gone through all of the debugging you can by yourself, don't forget that Heroku provides support for applications, in addition to the community-based forum support on Stack Overflow.

Even with support from outside sources, you will still need to troubleshoot your issue. Heroku can help you with its platform, but what about when the issue is inside of your app? The following debugging sections can help you solve your issue, or at least give you greater insight into what is happening with your app on the platform, which will help the support process go much smoother.

Let's take a look at some common deployment problems.

Deploy Debugging

I can't deploy, why?

As you use the Heroku platform, you will likely find yourself in this position at some time. Not to worry, though—deployment is the best place your application could fail. Ideally, it means that you just avoided deploying an app with a problem into production, which is great. Though sometimes it can be a bit difficult to understand where to get started debugging your deployment problems, here is a quick list of good places to begin:

- Check the status site.
- Reproduce the problem in another app.
- Copy config, add-ons, and labs features.
- Check the *.slugignore* file.
- Fork and use a custom buildpack with added logging statements.

Heroku Status Site

It sounds like an RTFM type of comment, but checking *http://status.heroku.com* can save you a bit of a headache if the issue isn't on your end. Heroku splits up its application architecture into two categories. The production category is everything that your "production quality" application needs to stay alive. This includes the routers, dyno capacity, and production tier databases. Noncritical components are grouped into "Development," which includes the API, command-line access, shared/development databases, and anything else that might go down that shouldn't affect your running app. By isolating failures to individual services you can help better understand the impact to your app. If development is down, you might not be able to push, but your old version of the app should still be able to run.

What happens if development is down on the status site? You can subscribe to notifications and wait for it to come back online. If everything is green and there aren't any other problems, you might want to try reproducing the issue on another app.

Reproducing the Problem

If the issue isn't clear by checking your deploy's output, sometimes you can isolate the failure by trying to reproduce it on another application. Hopefully you've got a staging server set up, and you can try deploying to that app. If everything works fine on your staging server but not on your production app, you should try to duplicate the production environment as closely as possible with your staging server. This includes making sure `config`, `addons`, and any labs features are on par across both servers:

```
$ heroku config
$ heroku addons:list
$ heroku labs:list
```

What do you do if there is a difference? You modify your staging server to have parity with your production server and then deploy again. If the last deploy worked, you might get an error saying that nothing changed because you haven't added a new commit in Git. A useful trick for forcing a deploy is to add a blank commit in your project:

```
$ git commit --allow-empty -m "debugging"
[master ddf2020] debugging
```

You should then be able to push and deploy. Once you've isolated the mechanism that causes your failure, you'll be better equipped to fix it. Reproducing the issue on staging or a fresh app is crucial if you believe you're getting your error as a result of the Heroku platform. You can try isolating the functionality that broke between deploys and adding it to a dummy example app.

Sometimes, after all of that, you still might not know where the failure is coming from. Then what can you do?

Check for a .slugignore File

A *.slugignore* file works much like a *.gitignore* file, except it tells Heroku what files to ignore when building your application. This can be pretty useful for a small subset of applications but it can also make some pretty hard-to-reproduce issues. Verify that you don't have one in your project; you probably don't, but it can't hurt to check.

So if you still don't know exactly what is causing your deploy issue, you can pull out the big guns and use a custom debugging fork of the buildpack.

Fork and Use a Custom Buildpack

Buildpacks are what Heroku uses to provide the step-by-step instructions that set up the environment for your app so it can run on a dyno. (Buildpacks are discussed further in Chapter 8.) The buildpacks are open source, so you can fork a buildpack, add debugging statements, and then deploy using that buildpack. If you haven't read Chapter 8, you should do so before attempting this debugging option.

Sometimes very difficult deployment issues become very obvious if you can isolate the problem to buildpacks. On one occasion an encoding issue in client code caused an exception while running the buildpack, causing it to not give any output at all. Once the developer deployed with a custom buildpack and just a few debugging lines, the trouble area was isolated and the problem became apparent.

How do you output from a buildpack? In the `compile` step of a buildpack, anything sent to the standard output will display in the build process. Depending on your language, you can add `puts`, `sprintf`, `System.out.println`, or `print()`, and the output will be available while building. But what to print? First, we'll figure out where our target problem area is with tracers.

Tracer bullets were first used by the British in 1915. The rounds have a pyrotechnic charge that burns brightly so you can see where the projectile goes, allowing the shooter to correct his aim. In programming, debug tracers are similar: they help us decode what is going on in our app. If you are trying to see if a particular piece of code gets executed, you can add a simple tracer to it:

```
print "=================="
```

If you have many such tracers, it can be useful to name them:

```
print "==== Detecting App"
```

Once you've isolated the problem area with tracers, you can dig in by outputting variable values or other debugging information.

OK, so we can use tracers output to discover which part of the build process is failing, but how do we actually add them?

To add your debug statements, first you fork the buildpack you are using. You can find them at *https://github.com/heroku*; just search for "buildpack." Once you've forked to your personal account such as *github.com/schneems*, you'll need to point the `BUILDPACK_URL` of your staging app to your repo, then add your tracers, and deploy your staging app. If you don't see any output, check that your changes were pushed correctly to the repo specified in the `BUILDPACK_URL`.

Once you're done debugging with your custom buildpack, you can simply remove the `BUILDPACK_URL` from your staging app's configuration vars and you're good to go.

Deploy Taking Too Long?

Heroku is an opinionated platform, and one of the opinions is that deploys shouldn't take too long. This prevents deployments from hanging, and gives you incentive to decrease your deployment time. If you accidentally deploy a bug, you don't want to go through an hour-long deploy process to get it into production. As such, the deployments have a maximum time that they will run; if your application takes longer to deploy than that time, it will be canceled. Although the value of the timeout may change in the future, it will likely stay somewhere around the 10-minute mark. Several common things that occur during deployments can ramp up the compilation time exponentially.

If your application must process assets—JavaScript, CSS, and image files on deployment—this can take a large amount of time. Why would an application need to process these files? It might gzip them, minify them, or convert them from one format to another such as turning SCSS files into CSS. If your application does this type of work, there are a few different things you can do to speed up the process. First, you will want to remove any asset files that you aren't using. If you are transferring assets to a third-party provider such as S3 during the compilation, consider instead using a service that will lazily cache your assets instead, such as a CDN with customizable origin. If things get bad, you might

actually consider splitting your app into several smaller apps, which can help with many areas, not just deployment time.

The final option is to move asset compilation to your local machine or an intermediate machine and not do the compile on every deploy. This option works well for assets that change infrequently, but it should be considered a last resort.

Besides asset compilation, downloading dependencies and running setup tasks can take a long time on very large apps. In those situations, you should consider if all of that really belongs in one app, or rather in several smaller ones.

Once you've got any deploy problems sorted out, it's all smooth sailing until you hit an error in running code.

Runtime Error Detection and Debugging

Unlike deploy errors, issues that occur during runtime can go unnoticed for some time. It is important that as an application owner you take steps to increase the viability of your application. In this section we will talk about some proactive measures you can take to watch for errors, as well as the steps you need to take to debug the errors. There are two common steps to debugging runtime exceptions: detection, where we find the exception, and debugging, where we fix the exception.

If errors are happening in your application, you need to know. This goes beyond exception notifications, and total application visibility should be the goal. But why? Isn't it just good enough to detect when errors happen and fix them? As Wayne Gretzky once said, "A good hockey player plays where the puck is. A great hockey player plays where the puck is going to be." We're building software, not playing hockey, but the statement still applies. Instead of chasing exceptions, make it a goal to know everything about the application, then when an exception does happen it is simply an unexpected state that needs to be explored. There are a number of simple steps you can take to increase visibility for you and your team.

Deploy Visibility

Deployments are the single most important action that any developer can take on a production app. Deployments are how bugs get introduced and how they get fixed. In recent years, it has become popular to adopt a "continuous deployment" strategy, where a team deploys not just once a year, or a quarter, or month but rather any time they need to. During active development it is not uncommon for a team to deploy many times in one day. That just makes it all the more critical that your team knows when someone is deploying and what code is live on the server. Small teams can get away with turning around and shouting to the office "I'm deploying," while larger teams can sometimes set up elaborate mechanisms that automatically lock out deploys and roll back if an

exception threshold is passed. To help with notifying team members and building deploy-related applications, Heroku offers a Deploy Hooks add-on:

```
$ heroku addons:add deployhooks:email
```

The deploy hook add-on is free and can be used to send emails, ping external services such as Basecamp or Campfire, post in IRC, or even send a POST HTTP message to a URL of your choice, if you need more flexibility. What you do with the information is up to you, but it's always a good idea to manually test out an application after deployment, which can help catch gaps that your automated tests missed. You are testing your app, right?

Test Visibility

One of the core tenets of continuous delivery (of which continuous deployment is a part) is testing your application with an automated test suite. Automated tests are code that makes assertions about your application code. There are many different types of automated tests with different purposes, from integration tests that use your app like a real user to unit tests to help with writing individual classes and methods. The different types of tests all help to protect against regression (when old features are accidentally lost) and application exceptions (when errors or exceptional states are introduced to your application). Because tests are run before your code is deployed, it can tell you if an application is safe to deploy. In addition to running tests locally, it can be a good idea to set up a continuous integration server (CI server), or a production-like environment where tests can be run. Developers can set up their own CI server or use a hosted service such as Travis CI or Tddium:

```
$ heroku addons:add tddium:starter
```

In addition to ensuring that your tests are being run, CI servers act to raise test status visibility to you and your team. Many application owners have CI set to alert everyone on a team if a build failed, and can help keep track of when a failing test was introduced, and by which developer. Testing may not catch every production issue, but it's simple to set up and can help with refactoring, so the time expended in setting up a good test suite will surely be paid back. What about those issues that come not from regression or exceptions, but from a gradual slowdown or sudden breakage of a data store or third-party service due to using too much capacity?

Performance Visibility

Even when your code is healthy, your app might not be. Most applications on Heroku rely on services such as Heroku Postgres and other data stores. These data stores are provided in a tiered system, so that as an application grows and needs additional resources they can be provisioned. It is important to keep a health check of all the subsystems that your application is using. You should check documentation on individual

services; for example, there are queries you can run using Postgres to determine if you need to upgrade. It is a good idea to run performance metrics once or more a week to determine if your application's data stores are running out of capacity. Depending on how your application is architected, you may need to check these statistics more often.

When you've got all of that in place, you can still get errors in your deployed application. Where should you look first to find them?

Exception Visibility

When you encounter errors in production, you're not going to have a debug page; you'll need to find the exception in your logs and start debugging from there. If you're used to a VPS, you might be reaching for that SSH command right now, but you can't SSH into your live box, as each dyno is sandboxed. Instead, you can use the Heroku `logs` command:

```
$ heroku logs --tail
```

This will provide you with a stream of all the logs from each dyno that is running. That means if you are running 1 dyno or 100, all of the output will be visible through this command. The importance of using logs to debug your application cannot be overstated. Many log messages will direct you to the exact file and line number where the exception occurs, some will even tell you exactly what went wrong. Errors in the code that you deploy are inevitable, and so is checking your logs.

Logging Add-Ons

Heroku will store 1,500 lines of logs on a rolling basis; if you need more than that, you should consider using a logging add-on. Heroku has several logging add-ons that will allow you to archive and search your logs. At the time of this writing, Logentries, Loggly, and Papertrail are available to store and search your logs:

```
$ heroku addons:add papertrail:choklad
```

Most of the add-ons have a free-to-try tier, so you can see which one is right for your app. They all have different interfaces and limits, and some have special features such as alerts. If you wanted to, you could even build your own log archiver using a log drain for your app, as log drains are a feature of Heroku's Logplex architecture. You can attach an unbounded number of log drains to a Heroku application without impacting the performance of your Heroku app.

Sometimes it's not the errors you're hunting for in the logs; sometimes you just want the error right in front of you. For those times, you might consider coding up a special admin error page.

Admin error pages

If your application has a logged-in user state and there is a restricted admin flag on users, it can be helpful to use dynamic error pages.

When logged in as an admin, if you come across an error, the error page can show you the backtrace and exception, while non-admin users just get the normal error page. Here is an example from the site *http://www.hourschool.com*:


```
Couldn't find Course with ID=chunkybacon

Details
Params: {"action"=>"show", "controller"=>"courses", "id"=>"chunkybacon"}

Backtrace:
lib/active_record/relation/finder_methods.rb:304:in `find_one'
```

While useful for debugging reproducible errors, you could get the same info from the logs; this method just makes it a bit easier. There are many times when you won't be able to reproduce the error, or when errors will happen without you knowing. To help combat this problem, let's take a look at some exception notification mechanisms.

Exception Notification

Exceptions still happen while you're asleep, and while a logging add-on might allow you to find the details, how do you know a user is getting exceptions in the first place? Exception notification services either integrate with your application code or parse log-files to store and record when exceptions happen. Because many of them group related exceptions, it can be very useful for determining the issues that affect the most users. They can also be configured to send emails or other types of notifications, hence the name:

```
$ heroku addons:add newrelic:standard
```

Some of these services, like New Relic (see Figure 7-1), do much more than record exceptions. They can be used to record and report your application performance and a

number of other metrics. As was mentioned previously, having insight into the performance of your app is crucial.

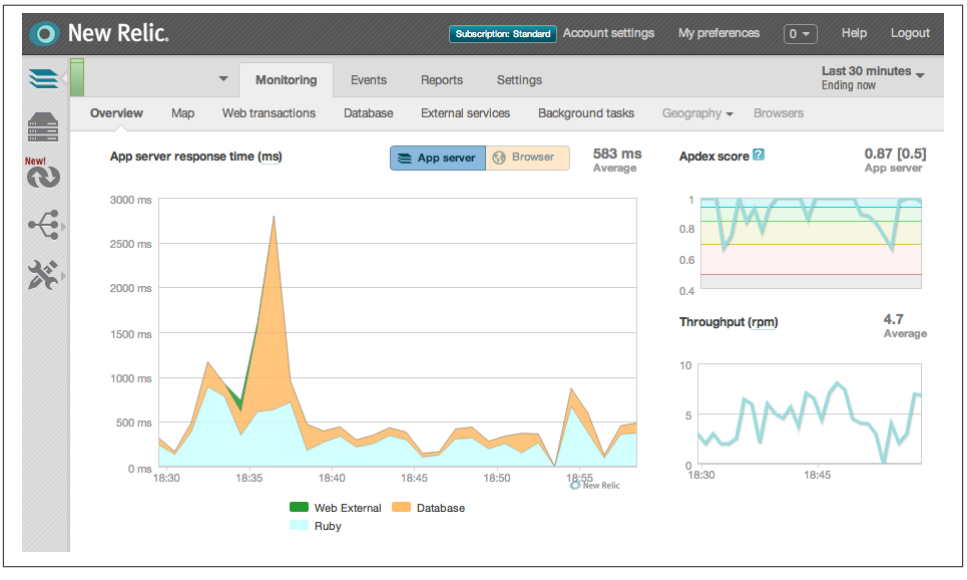


Figure 7-1. New Relic's web interface

Some of the add-ons have fewer features on purpose; they aim to have laser focus and try to minimize costs to the user by only providing what's needed. Shop around for an exception notification system that meets your needs.

Uptime Visibility

Uptime services that ping your app to check if it is still serving valid requests are used by many applications. These services will ping multiple public pages of your website at configurable intervals and alert you if they get any response other than a 200 (Success in HTTP). One popular service is Pingdom; it can be configured to send emails and text messages when an outage is detected. When you receive an outage report, you should always check *http://status.heroku.com* to confirm the outage is not part of a systemwide exception. If it is, you'll need to hunt down the exceptions using your logs or your notification services.

Twitter-Driven Development

So far, all of the techniques we have looked at to increase application visibility are technical in nature, but as the saying goes "the squeaky wheel gets the grease." Keep an ear open to social media channels such as Twitter, and maintain either an error-posting

forum such as Get Satisfaction, or at least a support email address. If your application has enough users, and you break something they want, they will let you know about the breakage. This should not replace any of the previously mentioned efforts, but it's always a good idea to be receptive to your users and their needs.

Code Reviews and a Branching Workflow

Because knowing more about your application is always beneficial, it can be a good idea to incorporate quick code reviews into your workflow. One popular option is by using Gitflow or GitHubFlow. In short, every developer works on a branch, then, instead of pushing to master, he submits a pull request where another developer must review the code. This doesn't guarantee better code, or fewer bugs, but it does guarantee that more developers are familiar with more of the new code going into production. What if someone breaks the production site on a Friday, but no one realizes it until Monday when that coder has gone on vacation? If the code was peer-reviewed, then at least one other developer is guaranteed to be somewhat familiar with the changes. Again, it's not always about preventing every possible runtime error, but rather knowing your application inside and out, so when disaster strikes, you are prepared to act swiftly and efficiently.

Runtime Error Debugging

Your system went down, and since you've got so much visibility into your application you were able to pinpoint the error and get a backtrace—now what? You need to understand what caused the issue in the first place so you can fix your production system. The first goal should always be to reproduce the problem, or at least the conditions in which the problems occur: if that fails, you will at least know what doesn't cause the error. In a perfect world, your error message would just tell you exactly what is wrong but what happens when you get a cryptic or misleading error message? It's time to put your debugging hat on—this is the fun part.

Data State Debugging

When you're getting an error in production but not in development, more often than not it's due to different states in your data store (such as Postgres). The most common of these is failure to migrate your database schema before pushing your code that uses new columns and tables. This problem was so common that Heroku's original stack, Aspen, automigrated your database for you. Unfortunately, large application owners need more control over their database migrations, and this behavior is not valid 100% of the time. Because of this, you need to make sure you run any migrations after you push them to your Heroku app. If you are using Ruby on Rails, your command might look something like this:

```
$ heroku run rake db:migrate
```

If you desire the automigration capacity, some developers build deploy scripts that can autorun migrations after every deploy. These scripts can be extended to do a number of other things, like running tests before deploying or checking CI status. Even if your database schema is correct, you can still experience unexpected scenarios due to the data inside of your database. To help solve this problem, it might be useful to clone a copy of your production data to a local machine. For more information on how to do this, refer back to Chapter 5.

Asset Debugging

Many modern web frameworks such as Ruby on Rails take a managed approach to dealing with assets. They allow a developer to write CoffeeScript instead of JavaScript or Sass instead of CSS, then generate files that can be understood by all browsers. If your framework does this type of task, the assets will be generated in the compile stage of deploying. A common error is writing these tasks to depend on configuration variables that are not present at compile time. Once deployed successfully, you may want to investigate the files that were generated. The easiest way to do this is with the `heroku run bash` command. This command will spin up a new dyno with your application loaded and give you access to the shell:

```
$ heroku run bash
Running `bash` attached to terminal... up, run.5026
~ $ ls public/assets -l
total 980
-rw------- 1 u31991 31991   3925 2013-10-29 16:41 manifest-
c928a583f4c6e55f59b889cfbac33539.json
# ...
```

From here you can use `cat`, `ls`, and `find` to debug low-level file generation issues.

Application State Debugging

If the error isn't associated with your data store, but rather with your code or your environment setup, then your first goal is to reproduce the issue. Here you have two options: you can reproduce the issue locally on your development machine or on a staging server that approximates your production server. If you cannot reproduce the issue locally, you will need a staging server, which is lucky because you can just spin up another Heroku app and try deploying to that server. Use additional information around the exception in addition to the backtrace and exception messages to help you reproduce the error. Check that you're using the same parameters in the problem request. Was the user logged in or out when she got the error? The closer you can get to the exact conditions under which the exception occurred the better chance you'll have of reproducing it.

Once you've figured out where the error is and how to reproduce it, you'll need to fix the underlying issue. Hopefully you can write a test case that covers the problem, or at

least reproduce it in an interactive console. Once you've gotten this far, it's just a matter of patching your code, deploying a fix, and then celebrating. In these scenarios, it can be helpful to have a retrospective plan in place, such as the 5 Whys, to see what caused the exception and if measures can be put in place to prevent a similar exception from happening in the future. However, it's important to remember that no system or code is ever 100% bug free—even systems on the space shuttle malfunction from time to time. The important thing is that when things do break, you understand how, and can hopefully reduce the impact of the exceptions.

You've got your site deployed and working like a charm. Remember that application visibility is a never-ending process. If you and your team stay diligent, you'll be able to deliver a world-class service that your users are happy to recommend to others. If you're not quite there yet, don't worry—there is always room for improvement. Read back over the suggestions in this chapter and see what your team could benefit from implementing.

CHAPTER 8
Buildpacks

Buildpacks provide the magic and flexibility that make running your apps on Heroku so easy. When you push your code, the buildpack is the component that is responsible for setting up your environment so that your application can run. The buildpack can install dependencies, customize software, manipulate assets, and do anything else required to run your application. Heroku didn't always have buildpacks; they're a new component that came with the Cedar stack.

To better understand why the buildpack system is so useful, let's take a look at a time before buildpacks. Let's take a look at the original Aspen stack.

Before Buildpacks

When Heroku first launched, its platform ran a stack called Aspen. The Aspen stack only ran Ruby code—version 1.8.6 to be exact. It had a read-only filesystem (no writes allowed). It had almost every publicly available Ruby dependency, known as gems, preinstalled, and it only supported Ruby on Rails applications.

Developers quickly fell in love with the Heroku workflow provided by Aspen and wanted to start using the platform for other things. The Ruby community saw Rack-based applications grow in popularity, especially Sinatra, and there was also an explosion in the number of community-built libraries being released. If you were a Ruby developer during this time you could be sure to find an *acts_as* library for whatever you wanted. While that was good for the community, keeping a copy of every gem on Aspen wasn't maintainable or sane, especially when different gems started requiring specific versions of other gems to work correctly. Heroku needed a way to deploy different types of Ruby applications and a different way to manage external library dependencies.

Recognizing the need for a more flexible system, Heroku released their next stack, called Bamboo. It had very few gems installed by default and instead favored declaring dependencies in a *.gems* file that you placed in the root of your code repository (this was

before Ruby's now-common dependency management system, called bundler, or the Gemfile, used to declare dependencies, existed). The Bamboo stack had just about everything a Ruby developer could want, but it didn't easily allow for custom binaries to be used, and it certainly didn't allow non-Ruby developers to take advantage of the Heroku infrastructure and workflow. The need for flexibility and extensibility drove Heroku to release their third and most recent stack, called Cedar. This stack was the first to support the buildpack system.

Introducing the Buildpack

With the Cedar stack, the knowledge gleaned preparing innumerable Ruby apps to run on Aspen and Bamboo was abstracted out into a separate system called a buildpack. This system was kept separate from the rest of the platform so that it could be quickly iterated as the needs of individual languages grew. Buildpacks act as a clean interface between the runtimes, which is the system that actually run the apps, and your application code. Among others, Heroku now supports buildpacks for Ruby, Java, Python, Grails, Clojure, and NodeJS. The buildpack system is open source, so anyone can fork and customize an existing buildpack. Developers might want to do this to add extra functionality such as a native CouchDB driver, or to install a compiled library like `wkhtmltopdf`. With a forked buildpack, you can do just about anything you want on a Heroku instance.

Even if you're not interested in building and maintaining a buildpack of your own, understanding how they work and how they are architected can be vital in understanding the deploy process on Heroku. Once you push your code up to Heroku, the system will either grab the buildpack you have listed under the config var `BUILDPACK_URL` or it will cycle through all of the officially supported buildpacks to find one that it detects can correctly build your code. Once the detect phase returns, the system then calls a `compile` method to get your code production ready, followed by a `release` method to finalize the deployment.

Let's take a look at each of these steps in turn.

Detect

When Heroku calls the `detect` command on a buildpack, it runs against the source code you push. The `detect` method is where a Ruby buildpack could check for existence of a Gemfile or a *config.ru* to check that it can actually run a given project. A Python buildpack might look for *.py* and a Java buildpack might look for the existence of *.mvn* files. If the buildpack detects that it can build the project, it returns a `0` to Heroku; this is the UNIX way of communicating "everything ran as expected." If the buildpack does not find the files it needs, it returns a nonzero status, usually `1`. If a buildpack returns a nonzero result, Heroku cancels the deploy process.

Once a buildpack detect phase returns a `0` (which tells the system that it can be run), the compile step will be called.

Compile

The compile step is where the majority of the build process takes place. Once the application type has been detected, different setup scripts can be run to configure the system, such as installing binaries or processing assets. One example of this is running `rake assets:precompile` in a recent Rails project, which will turn SASS-based styles into universal CSS, and CoffeeScript files into universal JavaScript.

In the compile stage, the configuration environment variables are not available. A well-architected app should compile the same regardless of configuration.

A cache directory is provided during the compile stage, and anything put in here will be available between deploys. The cache directory can be used to speed up future deploys (e.g., by storing downloaded items such as external dependencies, so they won't need to be downloaded again, which can be time consuming or prone to occasional failure).

Binary files that have been prebuilt against the Heroku runtime can be downloaded in this stage. That is how the Java buildpack supports multiple versions of Java, and how the Ruby buildpack supports multiple versions of Ruby. Any code that needs to be compiled and run as a binary should be done in advance and made available on a publicly available source such as Amazon's S3 service.

Binary Crash Course

On NIX-based systems, when you type in a command such as `cat`, `cd`, or `ls`, you are executing a binary program stored on the disk. But how does your operating system know where to find these programs?

You can find the location of commands by using `which`. For example, to get the location of the `cat` command, we could run:

```
$ which cat
/bin/cat
```

Here we see that the binary is in the path */bin/cat*. Instead of having the operating system look for our binary, we can run it from the full path if we desire:

```
$ /bin/cat /usr/share/dict/words
```

This will use the `cat` command, which can output one or more concatenated files. Many things you type on the command line are actually binaries that have been compiled to

run on your operating system; from `echo` to `ruby` to `python`, they are all just compiled files that you can open from anywhere on your system. But you don't need to type in the full path every time you execute a binary. How does the operating system know where to find them?

In the example, we executed `cat` by using the full path */bin/cat*, but we don't have to do that all the time; instead, we can just type in:

```
$ cat /usr/share/dict/words
```

How does our operating system know where to find the executable `cat` binary? It turns out that it searches all of the directories in the `PATH` environment variable in turn until it finds an executable file with the name you just typed in. You can see all of the paths that your operating system will search by running this command:

```
$ echo $PATH
/bin:/usr/local/bin:/usr/local/sbin:~/bin:/usr/bin:/usr/sbin:/sbin
```

Note that your `PATH` will likely look different than this. Here we have several paths such as */bin* and */usr/local/bin* in our `PATH` variable separated by colons. The dollar sign in the previous command simply tells our shell to evaluate the `PATH` variable and output that value.

Together, your `PATH` and the binary files on your computer make for a very flexible and useful system. You're not stuck with the binaries that were put on your system for you; you can compile new ones, add them to your `PATH`, and use them anywhere on your system. This concept is core to the philosophy behind UNIX as well as Heroku and buildpacks. The systems start out as bare shells, and are filled with the components needed to run your web app in the compile stage. You can then modify your `PATH`s and make those binaries available across the whole application.

Once the compilation phase is complete, the release phase will be called. This pairs the read-built application with the configuration required to run it in production.

Release

The release stage is when the compiled app gets paired with environment variables and executed. No changes to disk should be made here, only changes to environment variables. Heroku expects the return in YAML format with three keys: `addons` if there are any default add-ons; `config_vars`, which supplies a default set of configuration environment variables; and `default_process_types`, which will tell Heroku what command to run by default (i.e., `web`).

One of the most important values a buildpack may need to set is the `PATH` config var. The values passed in these three keys are all considered defaults (i.e., they will not

overwrite any existing values in the application). Here is an example of YAML output a buildpack might output:

```
---
addons:
config_vars:
  PATH: $PATH:/app/vendor/mruby_bin
default_process_types:
```

This YAML output will only set default environment variables; if you need to overwrite them, you need to use a profile script.

Once the release phase is through, your application is deployed and ready to run any processes declared in the Procfile.

Profile.d Scripts

The release phase of the build process allows you to set default environment variables, referred to by Heroku as config, on your application. While this functionality is useful it does not allow you to overwrite an existing variable in the build process. To accomplish this, you can use a *profile.d* directory, which can contain one or more scripts that can change environment variables.

For instance, if you had a custom directory inside of your application named *foo/* that contained a binary that you wished to be executed in your application, you could prepend it to the front of your path by creating a *foo.sh* file in *./profile.d/* so it would reside in *./profile.d/foo.sh* and could contain a path export like this:

```
export PATH="$HOME/foo:$PATH"
```

If you've written this file correctly in the compile stage of your build process, then after you deploy, your `foo` will appear in your `PATH`.

You can get more information on these three steps and more through the buildpack documentation.

Now you can detect, compile, release, and even configure your environment with *profile.d* scripts. While most applications only need functionality provided in the default buildpacks, you can extend them without needing to fork and maintain your own custom buildpack. Instead, you can use multiple buildpacks with your application.

Leveraging Multiple Buildpacks

Buildpacks give you the ability to do just about anything you want on top of Heroku's platform. Unfortunately, using a custom buildpack means that you're giving up having

someone else worry about your problems and you're taking on the responsibility of making sure your application compiles and deploys correctly instead of having Heroku take care of it for you. Is there a way to use Heroku's maintained buildpack, but to also add custom components? Of course there is: you can use a custom buildpack called Multi, to run multiple buildpacks in a row. Let's take a look at one way you might use it.

In a traditional deployment setup, you would have to manually install binaries (see "Binary Crash Course" on page 79) like Ruby or Java just to be able to deploy. Luckily, Heroku's default buildpacks will take care of most components our systems need, but it's not unreasonable to imagine a scenario that would require a custom binary such as whtmltopdf, a command-line tool for converting HTML into PDFs. In these scenarios, how do we get the code we need on our Heroku applications?

You'll need to compile the program you want so it can run on Heroku. You can find more information on how to do this in Heroku's developer center. At the time of writing, the best way to compile binaries for Heroku systems is to use the Vulcan or Anvil library.

Once you've got the binary, you could fork a buildpack and add some custom code that installs the binary for you. Instead, we recommend creating a lightweight buildpack that only installs that binary. Once you've got this simple buildpack, you can leverage the existing Heroku-maintained buildpacks along with another community-maintained buildpack called "heroku-buildpack-multi." This "Multi Buildpack" is a meta buildpack that runs an arbitrary number of buildpacks. To use it, first set the `BUILDPACK_URL` in your application:

```
$ heroku config:add BUILDPACK_URL=https://github.com/ddollar
                  /heroku-buildpack-multi.git
```

Instead of deploying using someone else's buildpack from GitHub, you should fork it and deploy using your copy. This prevents them from making breaking changes or deleting the code. A custom buildpack needs to be fetched on every deploy, so someone deleting his repository on GitHub could mean that you can't deploy.

Once you've got the `BUILDPACK_URL` config set properly, make a new file called *.buildpacks* and add the URLs to your custom buildpack and the Heroku-maintained buildpack.

You can see the documentation on Multi Buildpacks for examples and more options.

Quick and Dirty Binaries in Your App

If making your own mini buildpack seems like too much work, you can compile a binary and include it in your application's repository. You can then manually change the `PATH` config variable to include that directory, and you're good to go. While this process is simpler, it has a few drawbacks. It increases the size of your repository, which means it is slower to move around on a network. It hardcodes the dependency into your codebase, which can litter your primary application source control with nonrelated binary commits. It requires manually setting a `PATH` on new apps, which must be done for each new app, and it makes the binary not easily reusable for multiple apps. With these limitations in mind, it's up to you to pick the most appropriate solution.

We recommend using the multi buildpack approach when possible.

The Buildpack Recap

The buildpack is a low-level primitive on the Heroku platform, developed over years of running applications and three separate platform stack releases. It gives you the ability to have fine-grained control over how your application is compiled to be run. If you need to execute something that Heroku doesn't support yet, a custom buildpack is a great place to start looking. Remember, though, that an application's config variables are not available to the buildpack at compile time. It's easy to forget, so don't say you weren't warned.

Most applications won't ever need to use a custom buildpack, but understanding how the system works and having the background to utilize them if you need to is invaluable.

About the Authors

Neil Middleton has been developing web applications for 16 years across a variety of industries and technologies. Now working for Heroku based in the United Kingdom, Neil primarily spends his time helping Heroku's customers and training them in Heroku's use. Neil is a massive fan of keeping things simple and straightforward.

Richard Schneeman has been writing Ruby on Rails apps since version 0.9. He works for Heroku on the Ruby Task Force and is responsible for the Ruby buildpack. He teaches Ruby at the University of Texas. Richard loves elegant solutions and beautiful code.

Colophon

The animal on the cover of *Heroku: Up and Running* is a Japanese Waxwing (*Bombycilla japonica*), a fairly small passerine bird of the waxwing family found in Russia and northeast Asia.

The Japanese Waxwing is about 18 cm in length and its plumage is mostly pinkish-brown. It has a pointed crest, a black throat, a black stripe through the eye, a pale yellow center to the belly, and a black tail with a red tip. Unlike the other species of waxwing, it lacks the row of waxy red feathertips on the wing, which gives the birds its name.

The Japanese Waxwing's call is a high-pitched trill, and it feeds mainly on fruit and berries and also eats insects during the summer. There is little available information about breeding and nesting behavior—its courtship displays are probably similar to those of the other waxwings, performed with raised crest and fluffed gray rump feathers.

Changes in its habitat, use of pesticides, and other control measures from commercial fruit-growers have caused declines in population. This species is currently considered "near threatened" due to loss and degradation of its forest habitat.

The cover image is from Wood's *Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.