OPTIMIZING HEROKU APPLICATIONS

Expert Tips, Tools, and Scenarios for Managing Production Applications.

by Phil Ripperger

Contents

Preface ... 3

Section One - Common Issues ... 7

Chapter 1 - Where to Begin ... 8

Chapter 2 - SSL ... 12

Chapter 3 - The R14 & R15 Error Codes ... 24

Chapter 4 - The H12 Error Code ... 29

Section Two - Scenarios ... 40

Chapter 5 - The Magic of Heroku Postgres ... 41

Chapter 6 - Using New Relic Effectively ... 52

Chapter 7 - Customizing Heroku ... 61

Conclusion ... 68

Afterward ... 70

PREFACE

I love talking with developers who run their applications on Heroku. I enjoy answering questions about Heroku and providing tips or advice when I meet developers at conferences, user groups, or meet-ups.

I've supported web applications in one form or another for the past ten years. On some, I've been the developer and maintainer, but the majority of the applications I've seen have been in the official role of support. During the last few years as a support engineer at Heroku, I've debugged or analyzed over five thousand different applications. As such, I'm often able to look at an application having issues on Heroku and know what the problem is in under a minute.

Also, I would consider myself to have a somewhat-unique perspective of Heroku. I wasn't a founder of Heroku nor was I directly involved in the engineering, but I have an expert understanding of how Heroku works as well as many thousands of near-daily interactions with Heroku customers and their applications over the past few years. In short, I understand Heroku from the inside and out.

During my time supporting and debugging Heroku applications, a common set of issues and questions has become all-too familiar. We've done our best at Heroku to simplify and clarify these issues and questions but, given their nature, they will continue to be common. This guide will explain and show how to debug these common issues you will likely encounter while running an application on Heroku. This is not an official Heroku guide, rather it contains my personal perspective and thoughts on running production applications on Heroku, based on years of experience. My goal with this guide is to answer questions and provide advice as if you and I were to talk over a meal or drinks.

Some people claim that Heroku is a "black box" - I strongly disagree. At a basic level, Heroku is Unix. There's nothing magical about debugging a Heroku application and, as you'll see, the tools and methods I use to debug applications are the same tools and methods you have access to as a Heroku customer.

Who is this guide for?

If you have a production application running on Heroku or if you maintain more than one application, this guide is for you. Building and deploying an application is often the easiest part in the lifecycle of a Heroku application. But what happens when your social coupon site running on Heroku gets featured in TechCrunch? Or when mysterious timeouts seem to occur randomly for your application's users? Perhaps you've carefully designed and built the perfect iOS app, but didn't pay too much attention to the API application and database running on Heroku. Now the iOS app is gaining users quickly and you're not sure how to scale the parts running on Heroku.

I've written this guide for all of the above scenarios as well as common issues encountered while running applications on Heroku. This can be used as a troubleshooting guide, but hopefully reading through these tips, tools, and scenarios will help you avoid some common mistakes even when you deploy new applications in the future.

You might be new to deploying and supporting applications on Heroku, or you might be a Heroku pro with many long-running applications. My hope is that both new and seasoned Heroku customers will at least pick up some time-saving tips and learn something new.

Heroku has a great source of service information called the Dev Center. I highly recommend reading as many Dev Center articles as you can. This guide references the Dev Center, and builds on the basic information found there. Links to Dev Center content are provided where appropriate.

Finally, in order to get the most out of this guide, you should already be familiar with deploying applications on Heroku as well as have the Heroku Toolbelt installed. You will also need to be comfortable running commands in the terminal. If you're not familiar with basic Heroku concepts or comfortable with deploying an application via the command line, I would suggest reading this excellent Dev Center article on how Heroku works.

Disclaimer

Although I am still employed by Heroku, no part of this guide is official documentation. Again, this guide represents my personal experience and opinions on how to approach the issues. None of what I've written in this guide is secret information, and all of the tips, tools, and processes are freely available, whether in the Heroku Dev Center or elsewhere in the community.

With that taken care of, let's get started!

SECTION ONE

Common Issues

CHAPTER 1

Where to Begin

When debugging an application experiencing problems on Heroku, I nearly always start with one thing: the logs.

```
$ heroku logs -t
```

The logs command should output lines similar to the one below:

```
2013-05-08T00:28:23.205391+00:00 heroku[router]: at=info method=GET path=/ \
host=help.heroku.com fwd="210.149.23.47" dyno=web.1 connect=2ms service=111ms \
status=200 bytes=85
```

There's a lot of information in the line above. Throughout this guide I'll be showing sample log data. To keep the examples readable, I'll often truncate information we're not concerned about. For example, if we were interested in the `service` time in the log line above, I would use the following as a more readable example:

```
heroku[router]: … fwd="210.149.23.47" dyno=web.1 connect=2ms service=111ms status=200
```

I like to run the `heroku logs -t` command in a terminal window and keep it open while I check other aspects of the application. The `-t` flag will keep the command running and allow you to follow along as the logs update. This is known as 'tailing the logs' and it's very useful for catching ongoing exceptions, watching a deploy, checking for debug messages from the application, and generally keeping an eye on the application.

In some cases, it may help to filter the logs using the `-p` flag, for process type:

```
$ heroku logs -t -p router
```

For example, the above will show only the log stream from the router. The process filter works on any value inside the brackets (`heroku[router]` in the above example) after the timestamp.

It's also possible to filter on the source, which uses the `-s` flag and is based on the value outside the brackets:

```
$ heroku logs -t -s heroku
```

The above command will show only events in the log stream from `heroku[…]`, and none from the application itself.

Once the logs command is running, I look for certain indications of a problem. The most obvious is a stack trace from a dyno crashing. If I see a stack trace, debugging it locally is the next step. If no stack trace is present, I'll look for other indicators, such as Heroku-specific error messages like H12 or R14 (more on these later).

Logs are the most important part of debugging an app. So important that I recommend using a logging add-on such as Papertrail. Papertrail will store logs off-site which allows for later review. The reason this is useful is because the Heroku Logplex system provides a log stream for every application, but only keeps the last 1500 lines. Once the lines are replaced they are gone forever unless you have a logging add-on, or are storing the logs elsewhere using a syslog drain.

There is more information on logging, including syslog drains in the Dev Center logging article.

Next, if checking the logs does not reveal any immediate problem, I'll run a command to list the active processes:

```
$ heroku ps
```

This will show if processes are running or if they are in a crashed state.

```
$ heroku ps
=== web (1X): `bundle exec unicorn -p $PORT -c ./config/unicorn.rb`
web.1: up 2013/05/07 18:44:38 (~ 49m ago)
```

The above web dyno is running correctly. If it were crashed, the output would look like this:

```
$ heroku ps
=== web (1X): `bundle exec unicorn -p $PORT -c ./config/unicorn.rb`
web.1: crashed 2013/05/07 19:46:38 (~ 49m ago)
```

You might also see a corresponding line in the logs noting a dyno has crashed:

```
heroku[web.1]: State changed from starting to crashed
```

If a dyno is crashed, I will attempt to restart it and watch the logs to see if I can get a stack trace to look over:

```
$ heroku ps:restart web
```

The above command will restart all web dynos for the app. Since I'm tailing the logs, I'll be able to watch while the web dynos restart and, if they are crashing, I should find some clues (and hopefully a stack trace) in the logs.

Summary

As you can see, using the `heroku logs` and `heroku ps` commands are the easiest place to start when debugging or investigating an application running on Heroku. If there is something wrong with the application, looking over the output of both commands will usually provide enough clues to know where to look next.

In the following chapters, we'll look at some common problems that may show up for any application running on Heroku. While you work through the examples, be sure to keep a terminal window open and tail the logs of the application you are working with.

CHAPTER 2

SSL

SSL is a complex technology. It can be extremely confusing to set up your first SSL certificate, and sometimes, even when you know the ins-and-outs of SSL, it's still confusing. To make things worse, SSL is a technology that will sometimes appear to be working correctly, when subtle problems still actually exist and your application is not fully secure.

Heroku has done a lot behind-the-scenes to make the process of installing and maintaining an SSL certificate easier, but there are still plenty of confusing steps during the SSL process. For instance, you need a private key in order to get a new SSL certificate from a provider. But what's a private key, and how is one created?

Another confusing issue is that of the SSL provider versus Heroku: Heroku will host and serve an SSL certificate for your application; but you will need a third-party SSL provider to create and sign the actual SSL certificate for your application's domain(s).

Before we get into debugging SSL, let's start with the basic SSL process on Heroku.

SSL on Heroku

By default, all applications on Heroku have SSL support, provided you use the full Heroku application name:

```
https://application_name.herokuapp.com
```

This is possible because Heroku has a wildcard SSL certificate - *.herokuapp.com - that covers any subdomain (and thus application) running on Heroku.

Of course if you have a custom domain needing SSL, 'www.mydomain.com' for example, the above will not work. For SSL to be enabled for the custom domain attached to an application, you will first need to install the ssl:endpoint add-on. Here's the command:

```
$ heroku addons:add ssl:endpoint
```

At this point, you can either buy an SSL certificate, or use a self-signed certificate. If you're not ready to buy a new certificate, but need SSL for testing, I suggest a service called SelfSignedCertificate.com. Creating a self-signed certificate via this service is simple, and the resulting files will easily work with the `heroku certs:add` command.

If you're ready for a valid, trusted SSL certificate, there are a few steps you will need to take.

First, you will need at least three files:

1. A CSR - the Certificate Signing Request is a small file containing information about your application to be used by an SSL provider when creating the new SSL certificate.

2. A private key - this cryptographic key is used to verify both the CSR and the final SSL certificate.

3. A signed SSL certificate generated with both the CSR and private key. The creation of this file is what you pay an SSL provider for.

This may sound complex, but the end result is that you need the private key file and the new SSL certificate file to enable SSL on the ssl:endpoint add-on. Heroku has a good Dev Center article covering the entire SSL process here.

The Dev Center article outlines the process of creating a private key and a CSR via the `openssl` command. This process can be confusing if you're not familiar with the ins-and-outs of `openssl`.

If you're not familiar with `openssl` or comfortable generating your own private key and CSR, you can easily do so at CSRBuddy.com. CSRBuddy is a project I'm involved with, which was created after seeing too many Heroku customers struggle with the private key and CSR-generation process.

Once you have the CSR and private key files, you will then use the CSR when ordering a new SSL certificate from an SSL provider. I don't have any specific recommendations for an SSL provider, but most any provider will work with Heroku. There are two key things to remember during the process of

ordering an SSL certificate. First, have the CSR file handy, as you will either need to attach it to an order form, or paste the contents of the file into a form. Second, when asked, the server type needs to be either 'Nginx' or 'Apache'. Either server type will work with Heroku.

When you've completed the process of ordering an SSL certificate and have the new certificate file, you are ready to enable SSL support in your application:

```
$ heroku certs:add server.crt private_key.txt
```

Once the `heroku certs:add` command is successful, you can verify the correct setup via the following command:

```
$ heroku certs
Endpoint                  Common Name(s)                  Expires                 Trusted
------------------------  ------------------------------  --------------------  -------
hyogo-1003.herokussl.com  www.csrbuddy.com, csrbuddy.com  2013-06-06 23:59 UTC  True
```

Notice the value in the `Endpoint` column. You will need to make sure DNS for your custom domain points to this address. This can be accomplished by setting up a CNAME for your domain to the new endpoint address:

```
www.csrbuddy.com CNAME hyogo-1003.herokussl.com
```

It's worth noting that applications located in the Europe region will not have a distinct `herokussl.com` endpoint URL. For these applications, the endpoint URL will just be the `app_name.herokuapp.com` domain. The `heroku certs` command will reflect this difference.

Now that we've covered the basics, let's look at debugging a few common SSL issues.

DNS and the 'dig' command

In order for SSL to function, DNS for your custom domain and subdomains must be configured correctly. How can you check and verify DNS settings? There is a nice command line tool called `dig` that makes checking DNS configuration easy.

Here's a sample of the `dig` command for the above mentioned CSRBuddy.com application:

```
$ dig www.csrbuddy.com

; <<>> DiG 9.8.3-P1 <<>> www.csrbuddy.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 30790
;; flags: qr rd ra; QUERY: 1, ANSWER: 5, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.csrbuddy.com.         IN     A

;; ANSWER SECTION:
www.csrbuddy.com.   1800         IN     CNAME  hyogo-1003.herokussl.com.
hyogo-1003.herokussl.com. 150    IN     CNAME  elb003055-1983386325 ...
elb003055-1983386325.us-east-1.elb.amazonaws.com. 60 IN     A 50.19.111.122
elb003055-1983386325.us-east-1.elb.amazonaws.com. 60 IN     A 174.129.210.143
elb003055-1983386325.us-east-1.elb.amazonaws.com. 60 IN     A 107.22.234.193

;; Query time: 102 msec
;; SERVER: 192.168.1.1#53(192.168.1.1)
;; WHEN: Tue Jun  4 19:20:57 2013
;; MSG SIZE  rcvd: 176
```

The important part in the above output is the `;; ANSWER SECTION:`. You can see that `www.csrbuddy.com` is correctly configured as a CNAME to `hyogo-1003.herokussl.com`, which then resolves to a set of IP addresses.

Does the above output seem too verbose or confusing? You can use the following option to keep it simple:

```
$ dig +short www.csrbuddy.com
hyogo-1003.herokussl.com.
elb003055-1983386325.us-east-1.elb.amazonaws.com.
107.22.234.193
50.19.111.122
174.129.210.143
```

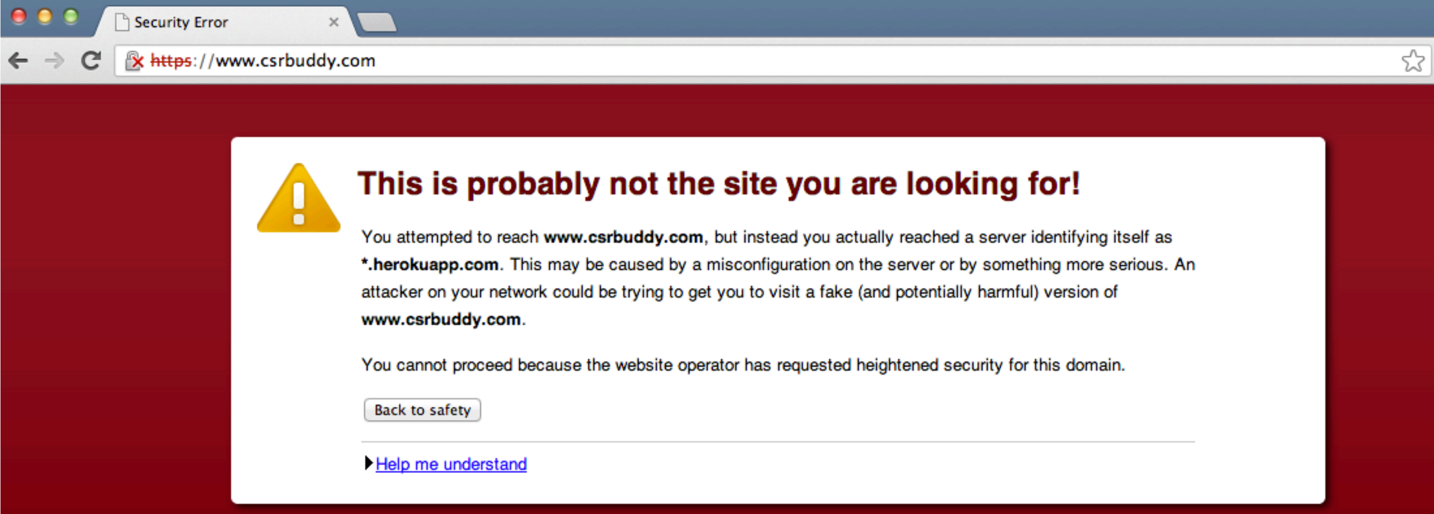
The important information is present, but nothing else.

The `dig` command is extremely valuable when it comes to checking and debugging SSL and DNS issues on Heroku. I make use of it nearly every day, and it's worth learning the basics of `dig`.

Serving the wrong certificate

One issue you may run into when setting up SSL is that the browser will report the certificate does not match the domain.

Perhaps you have just added a new SSL certificate via the `heroku certs:add` command and you are testing it, but getting this error in the browser:



What is causing this? The certificate appears to be correct, and is visible via the `heroku certs` command:

```
$ heroku certs
Endpoint                  Common Name(s)                  Expires               Trusted
------------------------  ------------------------------  --------------------  -------
hyogo-1003.herokussl.com  www.csrbuddy.com, csrbuddy.com  2013-06-06 23:59 UTC  True
```

Let's check the DNS settings again for this application with the `dig` command:

```
$ dig www.csrbuddy.com

; <<>> DiG 9.8.3-P1 <<>> www.csrbuddy.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52672
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.csrbuddy.com.        IN     A

;; ANSWER SECTION:
www.csrbuddy.com.   1751   IN     CNAME  csrbuddy.herokuapp.com.
csrbuddy.herokuapp.com.    11     IN     A      174.129.225.36
...
```

See how `www.csrbuddy.com` is still a CNAME to `csrbuddy.herokuapp.com`? This is why the Heroku wildcard certificate is being served and causing the browser to report an error. Once we update the DNS settings to point to the actual ssl:endpoint, everything will work:

```
$ heroku certs
Endpoint                  Common Name(s)                  Expires                 Trusted
------------------------  ------------------------------  --------------------    -------
hyogo-1003.herokussl.com  www.csrbuddy.com, csrbuddy.com  2013-06-06 23:59 UTC    True
```

Notice the 'Endpoint' value above is `hyogo-1003.herokussl.com`. We need to adjust DNS settings so `www.csrbuddy.com` is a CNAME to this endpoint:

```
$ dig +short www.csrbuddy.com
hyogo-1003.herokussl.com.
elb003055-1983386325.us-east-1.elb.amazonaws.com.
107.22.234.193
50.19.111.122
174.129.210.143
```

Working with an existing SSL Certificate

If your application is using an SSL certificate that has expired or will soon expire, you will need to update it. To check the current status of an SSL certificate, you can use either the `heroku certs` or `heroku certs:info` commands:

```
$ heroku certs:info
Fetching SSL Endpoint hyogo-1003.herokussl.com info for csrbuddy... done
Certificate details:
Common Name(s):     csrbuddy.com
                    www.csrbuddy.com

Expires At:     2013-06-06 23:59 UTC
Issuer:         /OU=Domain Control Validated/OU=PositiveSSL/CN=www.csrbuddy.com
Starts At:      2012-06-06 00:00 UTC
Subject:        /OU=Domain Control Validated/OU=PositiveSSL/CN=www.csrbuddy.com
SSL certificate is verified by a root authority.
```

In the above output, we can see the dates during which the certificate is valid. As it happened, while writing this chapter, I noticed the SSL certificate for CSRBuddy.com was nearing its expiration date. Here's how I updated the ssl:endpoint with a new certificate, using the `heroku certs` command.

The actual update command is easy enough: `heroku certs:update CERT KEY`

But what happens if you do not have access to the private key? Perhaps you lost or deleted it during the past year.

In that case, it's best to start over, generating a new private key and CSR yourself or using CSRBuddy.com. In fact, that's what I've done for the actual SSL certificate for CSRBuddy. I used CSRBuddy to generate its own CSR and private key. There's nothing wrong with starting the SSL process over. I believe it's more secure to rotate the private key each year, as opposed to re-using it. I submitted the new CSR to the SSL provider and received a new SSL certificate within a few minutes.

Once you have the new SSL certificate, it's a simple command to update the existing SSL certificate associated with the application:

```
$ heroku certs:update www_csrbuddy_com.crt private_key.key
Resolving trust chain... done
Updating SSL Endpoint hyogo-1003.herokussl.com info for csrbuddy... done
Updated certificate details:
Common Name(s): csrbuddy.com
                www.csrbuddy.com

Expires At:     2014-06-07 23:59 UTC
Issuer:         /OU=Domain Control Validated/OU=PositiveSSL/CN=www.csrbuddy.com
Starts At:      2013-06-04 00:00 UTC
Subject:        /OU=Domain Control Validated/OU=PositiveSSL/CN=www.csrbuddy.com
SSL certificate is verified by a root authority.
```

In the above output, notice the second line - 'Resolving trust chain... done'. This is actually a very nice feature that Heroku added to the SSL Endpoint add-on. Before, you would have had to worry about intermediate certificate chains, correct order, and multiple certificate files. Now, Heroku takes care of all that for you, so you only need a valid SSL certificate and matching private key. This one feature saved many hours of customer (and Heroku) frustration when implemented.

Verifying an SSL certificate is working

There are times where an SSL certificate is installed, the `heroku certs` command shows everything is correct, DNS settings are correct, yet the browser will still report something is wrong with an SSL certificate.

In this case, I usually use a third-party service called SSLShopper.com to check the certificate.

The SSL Checker from SSLShopper.com is a simple way to check the validity of an SSL certificate, including expiration date and valid intermediate certificates. Here's an example:

SSL Checker

This **SSL Checker** will help you diagnose problems with your SSL certificate installation. You can verify the SSL certificate on your web server to make sure it is correctly installed, valid, trusted and doesn't give any errors to any of your users. To use the SSL Checker, simply enter your server's hostname (must be public) in the box below and click the Check SSL button. If you need an SSL certificate, check out the SSL Wizard.

More Information About the SSL Checker

Server Hostname: www.csrbuddy.com **Check SSL**

(e.g. www.google.com)

www.csrbuddy.com resolves to 54.228.223.52

Server Type: thin 1.5.1 codename Straight Razor

The certificate should be trusted by all major web browsers (all the correct intermediate certificates are installed).

The certificate was issued by Comodo. **Write review of Comodo**

The hostname (www.csrbuddy.com) is correctly listed in the certificate.

This certificate will expire in 2 days. Renew now.

Common name: www.csrbuddy.com
SANs: www.csrbuddy.com, csrbuddy.com
Valid from June 5, 2012 to June 6, 2013
Serial Number: e15cd7bfd88503ab21a82dcea809bdf2
Signature Algorithm: sha1WithRSAEncryption
Issuer: PositiveSSL CA 2

Common name: PositiveSSL CA 2
Organization: COMODO CA Limited
Location: Salford, Greater Manchester, GB
Valid from February 15, 2012 to May 30, 2020
Serial Number: 076f124681459c28d548d697c40e001b
Signature Algorithm: sha1WithRSAEncryption
Issuer: AddTrust External CA Root

Common name: AddTrust External CA Root
Organization: AddTrust AB
Location: SE
Valid from May 30, 2000 to May 30, 2020
Serial Number: 1 (0x1)
Signature Algorithm: sha1WithRSAEncryption
Issuer: AddTrust External CA Root

Summary

Hopefully these tips and tools will provide clarity to the SSL process on Heroku. And the next time one of your SSL certificates expires, updating it won't be such a daunting task.

CHAPTER 3

The R14 & R15 Error Codes

If you see an R14 error in your application's logs, don't panic - you're not the only one! After H12, R14 is probably the most common error on Heroku. I like to think of R14 as a warning (though serious), and R15 as the actual error.

First, it's worth noting that each dyno running on Heroku is allocated a set of resources on the underlying instance - disk space, memory, and CPU. R14 and R15 errors specifically deal with RAM usage by a dyno. With the default 1X dyno, the memory limit is set at 512 MB of RAM. The larger 2X dyno is allocated 1024 MB of RAM, double that of the 1X dyno. You can read more about dyno sizes here.

With the above limits in mind, an R14 error is a warning from the Heroku dyno manager that a dyno is using more than its fair share of RAM. This causes the dyno to swap RAM to disk and severely degrades performance. Put simply, R14s directly impact the performance of an application by causing a dyno to run more slowly. This in turn can lead to other errors (such as an H12) if the dyno slows down too much.

Here's what an R14 will look like in your application's logs. Note the `mem=533M`:

```
heroku[web.1]: Process running mem=533M(104.3%)
heroku[web.1]: Error R14 (Memory quota exceeded)
```

In the above example, the `web.1` dyno is using 533 MB of RAM, which is 104.3% of the 512 MB limit.

Normally you will not need to worry about physical resources such as RAM on Heroku, but if an application has a memory leak or is written in such a way as to be memory-intensive, you will see the R14 error in the logs.

Once a dyno uses 300% of the available RAM, the dyno will automatically be killed via the SIGKILL signal and an R15 error will show up in the logs. The dyno will then be restarted.

This is a typical R15 error:

```
heroku[web.1]: Process running mem=2565MB(501.0%)
heroku[web.1]: Error R15 (Memory quota vastly exceeded)
heroku[web.1]: Stopping process with SIGKILL
heroku[web.1]: Process exited
```

R15 errors are particularly bad because the dyno is killed abruptly. Any processes running within the dyno are terminated immediately. In the case of a web dyno, this will often cause an error to be sent back to the client. If a worker dyno receives an R15, any work being done will be immediately stopped, which can lead to other problems.

Keep in mind that an R15 error is almost always related to a memory leak. With that in mind, what is the best way to debug R14 and R15 errors?

To start, it's easiest to run the application locally and monitor RAM usage of the local-running processes. If you suspect a certain action or path through the application causes RAM usage to spike, simulate that locally. By doing this, you should be able to determine which objects or methods are using large amounts of RAM and refactor code where necessary.

If you are using OS X to debug locally, the built-in OS X Activity Monitor is a good utility to track RAM usage. Boot your application locally, mirroring production as closely as you can (including using production data, if possible). Once running, search for the process type in Activity Monitor, as seen here:

Note the Ruby process running the local Rails application is using ~100 MB of RAM. If you notice this number increase over time, or when a specific action is processed, it will help identify where the memory leak is.

If Activity Monitor is not helping, the next step is to use a library such as Oink or Heapy. Both are language-specific, but the general idea is to output memory usage to the logs and analyze the information later. When using Oink or a similar tool, you will want to make use of a syslog drain or a logging add-on such as Papertrail, to capture log data for later review.

It may also help to step back and analyze how the application is using data. For instance, if you have a single worker dyno that is churning through a million records and constantly generating R14 errors, it's time to upgrade to 2X worker dynos or split the work amongst two or more workers. The same goes for web requests - loading too much data at once inside a single web request is a common cause of R14s. Pagination will help in this case.

The above steps are great for investigating a memory leak once you are aware of the problem, but what happens when the memory leak or underlying issue using too much RAM is more subtle? The culprit is often a complex bug or scenario that starts slowly but ultimately leads to increased RAM usage over a longer period of time. Because of this, it may help to track the dyno RAM usage over time using the `log-runtime-metrics` feature. This handy Heroku Labs feature will emit dyno resource usage to the application log stream:

```
heroku[web.1]: … measure=load_avg_15m val=0.16
heroku[web.1]: … measure=memory_total val=96.79 units=MB
heroku[web.1]: … measure=memory_cache val=0.01 units=MB
heroku[web.1]: … measure=load_avg_1m val=0.04
heroku[web.1]: … measure=memory_pgpgout val=103655 units=pages
heroku[web.1]: … measure=memory_rss val=96.77 units=MB
heroku[web.1]: … measure=load_avg_5m val=0.15
heroku[web.1]: … measure=memory_swap val=0.00 units=MB
heroku[web.1]: … measure=memory_pgpgin val=128432 units=pages
```

In the above output, the key value related to R14s and R15s is the `measure=memory_swap` which has a `val=0.00`, indicating no swap is being used by the dyno. If the value were to increase over time or suddenly, it would indicate the dyno is using too much RAM and is swapping to disk.

Once `log-runtime-metrics` is enabled, the log data can be analyzed and alerting can be set up for key values. Once again, I recommend the Papertrail logging add-on for this.

Summary

In general, the source of R14 and R15 errors can be difficult to track down. If you have not quickly found the cause of increased RAM usage, be prepared for an ongoing investigation into your application. If you get to this point, don’t rule out any part of your application. I’ve seen many plugins or third-party libraries turn out to be the source of an edge-case memory leak that happens over the course of twelve or more hours. Isolating this type of bug is tedious, but the above tools and information will help.

CHAPTER 4

The H12 Error Code

My guess is that you've probably seen or heard of the H12 error code. An H12, also frequently referred to as a timeout, is very common, simply because of the reason it occurs. The H12 error is the result of a simple Heroku convention: there is a 30-second limit on all web requests. This may sound arbitrary or too strict, but in reality any application that forces the user or client to wait more than a few seconds is too slow. A single, long-running request can cause an H12 timeout, but so can a series of shorter requests. In my opinion, a perfectly-tuned application would have all requests running in 500ms or less. However, not every application is perfectly tuned when first deployed to Heroku, so you are likely to see an occasional H12 for any application that handles a fair amount of traffic and has requests that run longer than a few seconds.

Before we dive into the details of an H12, it's worth noting that certain types of requests should not be handled within the normal request / response cycle of a web application running on Heroku:

- Image uploads, resizing, or processing
- Complex report generation
- Social graph calculations
- Third party API calls
- Sending email

The first item, image uploads, is a very common cause of H12s. Rails applications in particular make it easy to accept image uploads. If your application receives an image upload, resizes the image or otherwise manipulates it, then saves or moves the image elsewhere, it will at minimum take several seconds to process each upload. This type of long-running request will significantly impact the performance of the application, and will most likely cause H12 errors. For this reason, I want to note that image uploads and image processing need to be handled via a background worker dyno. In most cases, it's best to upload an image from the client directly to S3, then use a worker dyno to retrieve the image from S3, process it, and store the results. Building this process will require more work up-front, but it will save a lot of headaches later on as your application handles more and more traffic.

Details of an H12 Error

Now back to the H12 error. At first, it may be difficult to identify an H12. The reason for this is that when an H12 is triggered, an HTTP 503 (Gateway Timeout) error will be returned to the client, which by default shows up as an application error. After seeing the 'something went wrong' error page in your application, you may be tempted to search the logs and look for a full stack trace, which would indicate an exception at the application level. However, an H12 error is reported by the Routing layer and is easy to spot. It will look like this:

```
heroku[router]: at=error code=H12 desc="Request timeout" method=GET path=/coupons \
host=debugging.herokuapp.com fwd=52.171.38.92 dyno=web.2 queue= wait= connect= \
service=30000ms status=503 bytes=0
```

I should note that the default application error page is not something you want customers to see, and I recommend configuring a custom error page that will be shown when an H12 (or other Heroku specific error) occurs.

Now that you know what to look for, let's start with a very simple example of how an H12 can occur.

As I mentioned earlier, by convention Heroku has a 30-second limit on all web requests. Note that this limit is platform-wide and cannot be changed. An incoming request will first hit the Routing layer. The Heroku Routing layer is designed to be very fast and simple. When a request comes in, the Routing layer randomly selects a `web` dyno the application is running and passes the request to the dyno. At this point, the 30-second timer starts. If the Routing layer does not receive a response from the dyno within 30 seconds, the Routing layer cuts short the request and returns an HTTP 503 (Gateway Timeout) error code to the client. Meanwhile, the dyno (or process within the dyno for concurrent servers such as Unicorn) handling the request will continue to process until it completes the code path. This effectively ties up the dyno (or process) and it will be unable to handle additional requests until it completes. This introduces the possibility of requests queueing at the dyno level and additional H12s.

With this in mind, imagine a request to your application that triggers a customer's password to be reset, then an email is sent to the customer with a link to create a new password. If the service being used to send email is down or running slowly, this request will 'hang' when attempting to send the email, and will result in an H12 response after 30 seconds. The client will see an error, but depending on the actual problem with the email service, the email may still be delivered, as the dyno handling the request will continue processing.

This is obviously a simple example, but it should be clear that H12s can occur easily with a single long-running action.

The H12 scenario becomes increasingly complex as you add more traffic to the application, and more than one slow action. As traffic increases, and the number of slow actions increases, queueing at the dyno level occurs, and requests that are normally very fast suddenly start to result in H12s.

It's worth noting that the Heroku Routing layer is big enough and fast enough that very rarely do requests queue at the Routing layer itself. Queueing instead happens at the dyno level. And remember that the Routing layer is stateless and does not know which of the application's dynos are busy, or even if a particular dyno has any requests in its queue. The Routing layer simply and quickly handles incoming requests, then passes the request to a web dyno and awaits a response.

It's worth repeating: the Heroku Routing layer is random and WILL route requests to a busy dyno. The Routing layer does NOT know about any queue of requests at the dyno level. This may sound like a flaw in the Routing layer, but the Routing layer was purposefully designed and built to function this way.

Let's step back for a moment. There has been much discussion in the community about the Heroku Routing layer. Some would argue that Heroku's Routing needs to change, or is otherwise flawed. I disagree, but rather than get into the details and arguments here, I'll leave it to the reader to research and come to your own conclusion. I do recommend the following articles as a good place to start learning the details of Heroku's Routing architecture:

https://blog.heroku.com/archives/2013/4/3/routing_and_web_performance_on_heroku_a_faq

https://devcenter.heroku.com/articles/http-routing

http://aphyr.com/posts/278-timelike-2-everything-fails-all-the-time

Remember, once an H12 is triggered, the long-running request will continue to process, tying up the dyno or process and potentially causing additional requests sent to the dyno to queue. This situation is not so severe for application servers that are event-driven or have child processes - Node.js or Unicorn for example. However if a single process application server such as Thin is being used, there is no way to handle additional incoming requests save for a queue. Further H12s can happen simply because the requests waiting in the queue have their 30-second timer running too.

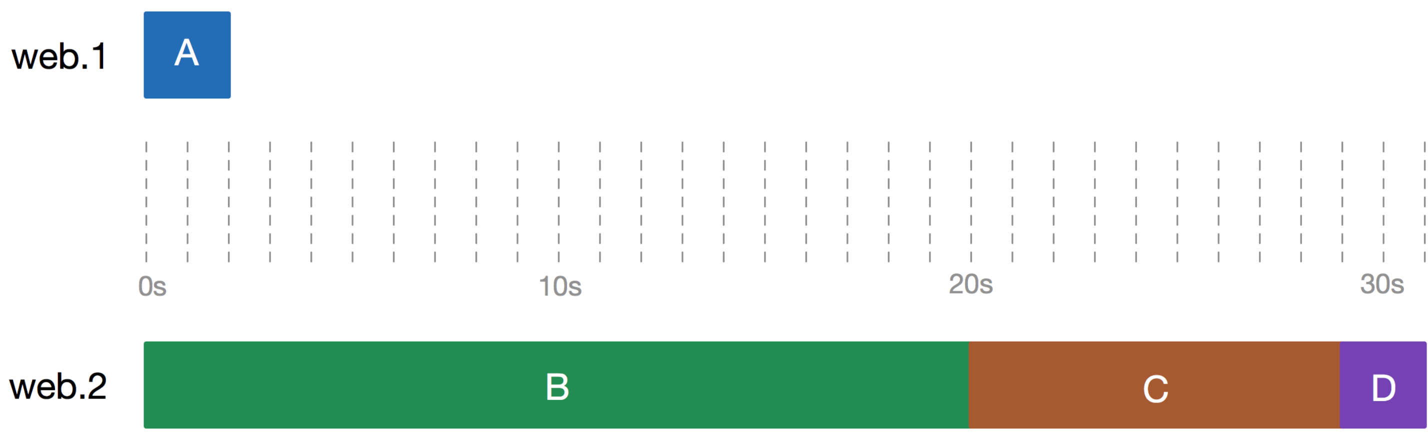
It’s worth noting here that a concurrent web server such as Unicorn or Puma for Ruby, Gunicorn for Python, or Node.js in general, will perform much better and be less susceptible to H12s and queueing because each web dyno will have more than one process to handle incoming requests. A single process web server, such as Thin, can only handle one request at-a-time and will see much more queueing.

Let's walk through a more complex example, using a non-concurrent application server, where two long-running requests will actually cause a 'fast' request to result in an H12:

- an application is running two web dynos, web.1 and web.2
- four requests arrive at the same time, requests A, B, C, and D

- request A takes 2 seconds to process
- request B takes 20 seconds to process
- request C takes 9 seconds to process
- request D takes 2 seconds to process

- request A is randomly routed to web.1
- requests B,C,D are randomly routed to web.2



In the above example, request A is handled normally, and the web.1 dyno is freed up after 2 seconds to handle additional requests. Meanwhile, requests B, C, and D were randomly routed to web.2, and requests C and D are queued at the dyno level of web.2, waiting on request B. Once request B finishes, request C is handled as it's next in the queue. Request C will process normally, but the end result is that request D will be cut short and an H12 will be returned, because request D will have waited for 29 seconds while requests B and C were processed. Request D will actually start processing, nearly finish, but will miss the 30-second cut-off at the Routing level. This is the danger of slow requests! Request D is normally very fast, but will have already been in the queue for 29 seconds before it gets a chance to be processed, and the end result is an H12 for the client that submitted request D.

Obviously this is a simple example, but you can see how slow requests, even slow requests much less than the full 30-second limit, can cause other queued requests to result in an H12. The more traffic and dynos an application has the more complex the scenarios become.

Rather than get into more and more complex scenarios, I'm going to leave it to the reader to imagine scenarios within their own application that could cause H12s. Remember that no application that runs web dynos is immune to H12s.

When it comes to live production applications on Heroku, I've seen the full range of H12 scenarios. From low-traffic applications encountering H12s because of a single slow action, to high-traffic applications handling thousands of requests per second seeing a spike in H12s because one frequently-requested action is running for 5 seconds instead of the normal 100ms. The most difficult H12 scenarios to diagnose happen where a busy application has a number of slow actions in the 5 to 10 second range, where all are causing H12s and each action must be investigated and optimized.

Finding the Cause of H12 Errors

Now that you know how to spot an H12 and how it occurs, how do you diagnose the cause? How do you determine which part of an app is running slow and causing H12s? The key is to find, then fix, every slow action in your application. This is an iterative process that will continue through the life of your application. As long as data, traffic, or code is changing in your application, you will need to watch for long-running actions. A good way to start tracking H12s is to setup alerts in the logging add-on used by your application.

Remember, if your application is sending email, processing payments, or doing other processing that takes more than five seconds, these actions need to be moved to a background dyno for asynchronous processing. Keeping these types of long-running tasks in the standard request / response cycle is asking for trouble.

If you have the appropriate tasks running in the background and are still seeing H12s, the next step is to limit the response time within the application using something like rack-timeout. The rack-timeout

library is Ruby-specific, but the same concept applies to other languages and frameworks. Limiting the response time and raising or logging an exception is the best way to identify long-running actions. Rack-timeout is a great example of how to approach limiting requests. Not only does it limit any request from taking longer than 15 seconds by default, it also provides extra information about the request. Once the long-running actions are caught and surfaced in the log stream via rack-timeout, you can optimize and fix them, which will reduce or eliminate H12s.

Let's look at the output of rack-timout more closely. Requests that take longer than the default 15 seconds will result in an exception, visible in the log stream:

```
app[web.1]: source=rack-timeout id=f40adf06fa2dd4137fd9265b0339981c age=329ms \
timeout=15000ms state=ready at=info
app[web.1]: Started GET "/coupons" for 173.0.4.85 at 2013-06-09 01:52:08 +0000
app[web.1]: source=rack-timeout id=f40adf06fa2dd4137fd9265b0339981c age=329ms \
timeout=15000ms duration=15009ms state=timed_out at=error
app[web.1]: Completed 500 Internal Server Error in 14984ms
app[web.1]:
app[web.1]: Rack::Timeout::RequestTimeoutError (Request ran for longer than 15 seconds.):
app[web.1]:   app/controllers/coupons_controller.rb:7:in `sleep'
app[web.1]:   app/controllers/coupons_controller.rb:7:in `index'
app[web.1]: source=rack-timeout id=f40adf06fa2dd4137fd9265b0339981c age=329ms \
timeout=15000ms duration=15013ms state=completed at=info
heroku[router]: at=info method=GET path=/coupons host=debugging.herokuapp.com \
id=f40adf06fa2dd4137fd9265b0339981c fwd="173.0.4.85" dyno=web.1 connect=2ms \
service=15024ms status=500 bytes=643
```

Notice in the above log data a `Rack::Timeout::RequestTimeoutError` exception is raised, and an HTTP 500 is sent back to the client. In this simple example we can see, via the second line, that the call to `/coupons` is resulting in an H12. If this were a real application, I would open the source for the `/coupons` action and look for reasons why it may be running slow. Once I've made optimizations or fixes, I would deploy the changes and watch the logs to see if H12s continue.

In general, I recommend setting rack-timeout's timeout value to 10 seconds or less, by overriding the default. It makes no sense to use a timeout value of 28 seconds. The lower the better. And once your app is tuned and every action is running fast, you can drop the timeout to 5 seconds or less.

```
# config/initializers/timeout.rb - in seconds
Rack::Timeout.timeout = 10
```

Also, there are other extremely useful pieces of information that rack-timeout provides. Take a look at the line below, which is actually the first line from the above example.

```
app[web.1]: source=rack-timeout id=f40adf06fa2dd4137fd9265b0339981c age=329ms \
timeout=15000ms state=ready at=info
```

The above line is what rack-timeout will first emit when it sees a request. The first thing to note is the `age=329ms` value. When a request first enters the Heroku Routing layer, it is given an HTTP header called `X-Request-Start`, which has a timestamp. When rack-timeout first sees a request, after it has passed through the Routing layer, makes it through the dyno-level queue, and finally reaches the application, it will calculate the total time between `X-Request-Start` and the current time. In this example, the request spent 329ms in the Routing layer and the dyno queue, before making it to the application. If this value were high, say 5000ms (5 seconds) or higher, you would know that requests are probably spending too long in the dyno-level queue.

The next line from rack-timeout is actually emitted when the request is interrupted at 15 seconds:

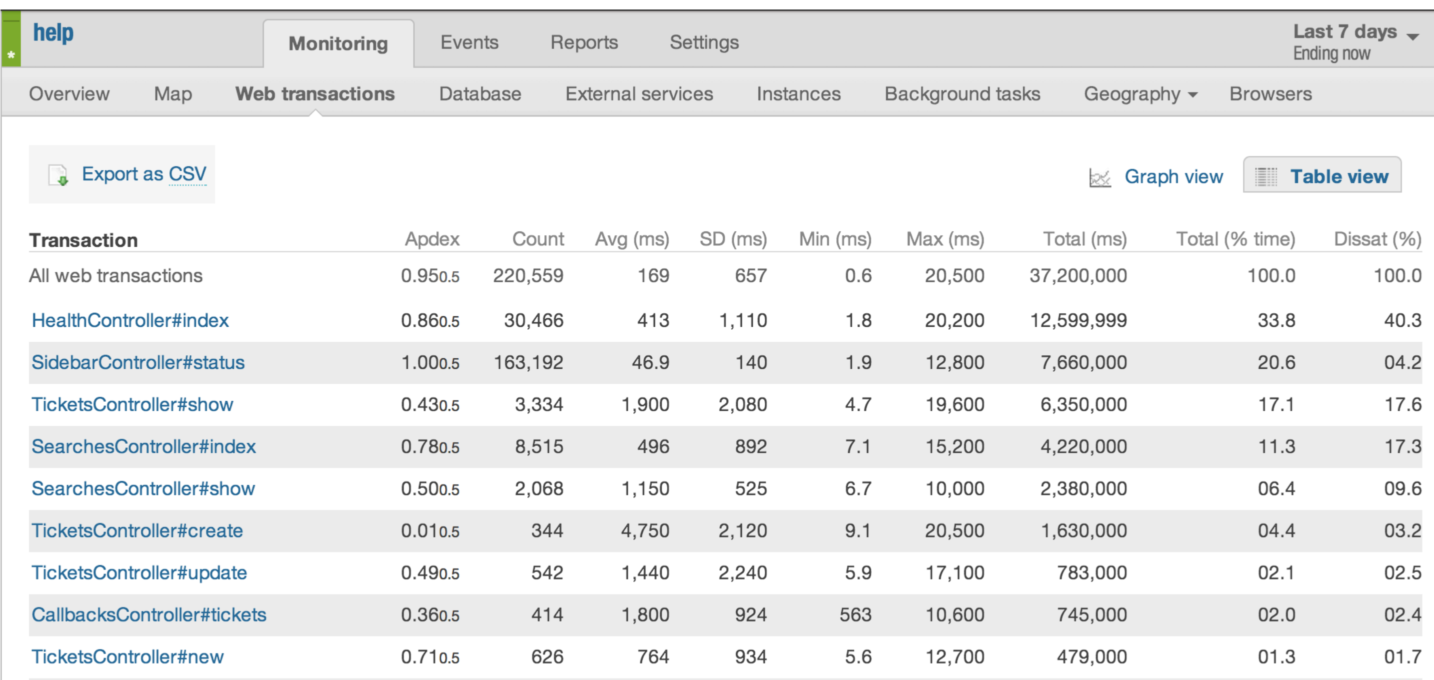
```
app[web.1]: source=rack-timeout id=f40adf06fa2dd4137fd9265b0339981c age=329ms \
timeout=15000ms duration=15009ms state=timed_out at=error
```

Note the `duration=15009ms state=timed_out` values, which indicate the request lasted just over 15 seconds and timed-out. When looking through log data, keep an eye out for requests with high `age=` values and low `duration=` values. This would indicate that fast-running requests are spending too much time in the queue. Remember, if a request that normally completes in 50ms spends 15 seconds in the dyno-level queue, it would have an `age=` value of at least 15,000ms. And keep in mind that with rack-timeout installed, requests that will result in an H12 even before leaving the dyno level queue are dropped by rack-timeout and not processed by the application.

Also, note the `id=xxxx` value rack-timeout emits. This value is unique for each request, and is useful for tracking or filtering requests when logs are extremely busy or verbose. In particular, this value can be set via the http-request-id feature currently in Heroku Labs.

Hopefully you can see that rack-timeout or a similar method will help to quickly pinpoint slow actions in your application. By tailing your application's logs or using an add-on such as [Papertrail](), you will be able to see which actions are encountering timeouts, determine the underlying issue, and fix.

One final tool that is useful for finding slow running actions that are the likely cause of H12s is New Relic. In particular, New Relic has a feature that lets you view statistics on actions in a table format. Here’s a screenshot of what it looks like, under the Monitoring / Web transactions tab, selecting the Table view option on the right.



Transaction	Apdex	Count	Avg (ms)	SD (ms)	Min (ms)	Max (ms)	Total (ms)	Total (% time)	Dissat (%)
All web transactions	0.95 0.5	220,559	169	657	0.6	20,500	37,200,000	100.0	100.0
HealthController#index	0.86 0.5	30,466	413	1,110	1.8	20,200	12,599,999	33.8	40.3
SidebarController#status	1.00 0.5	163,192	46.9	140	1.9	12,800	7,660,000	20.6	04.2
TicketsController#show	0.43 0.5	3,334	1,900	2,080	4.7	19,600	6,350,000	17.1	17.6
SearchesController#index	0.78 0.5	8,515	496	892	7.1	15,200	4,220,000	11.3	17.3
SearchesController#show	0.50 0.5	2,068	1,150	525	6.7	10,000	2,380,000	06.4	09.6
TicketsController#create	0.01 0.5	344	4,750	2,120	9.1	20,500	1,630,000	04.4	03.2
TicketsController#update	0.49 0.5	542	1,440	2,240	5.9	17,100	783,000	02.1	02.5
CallbacksController#tickets	0.36 0.5	414	1,800	924	563	10,600	745,000	02.0	02.4
TicketsController#new	0.71 0.5	626	764	934	5.6	12,700	479,000	01.3	01.7

In the above image, note the Max(ms) column. When combined with the Count column, it’s easy to see long-running actions that are being called frequently. If this application were experiencing H12 errors, I would start by looking at the HealthController#index action to see why it is running for more than 20 seconds.

Summary

Hopefully you can see that H12s are a complex topic. Sometimes there is an easy answer for why an application sees H12s, sometimes the answer is not so easy. But using tools like rack-timeout and New Relic, you can quickly identify possible causes and work to optimize an application. Keep in mind that finding, fixing, and monitoring for H12s is an on-going process. And always remember that slow responses are like poison to a Heroku application. They can have many side effects and even one can cripple an application with performance issues.

Here's the bottom line:

Do everything you can to keep slow requests out of your application.

SECTION TWO

Scenarios

CHAPTER 5

The Magic of Heroku Postgres

Heroku Postgres is one of the most amazing services I've ever used. I say that not just as an employee of Heroku, but as a former database administrator. I started with Access (yeah, I know), moved to SQL Server, then on to MySQL, dabbled in CouchDB and MongoDB, before settling down firmly with Postgres as my database of choice. The Heroku Postgres team operates one of the largest fleets of Postgres servers in the world, and it's growing quickly. Some of the utilities and optimizations developed by Heroku to manage these servers have actually made their way into the Postgres codebase. And hardly a week goes by where I'm not amazed at what the Heroku Postgres team is doing.

With the above praise out of the way, it can be difficult to keep track of all the helpful Postgres features available to you as a developer. Here are a few scenarios that make Heroku Postgres so amazing:

- Is your application suffering because of a slow or erratic database server? You can swap out the database server for something twice as big, four times as big, or larger, with minimal downtime.

- Sending an important email that will bring a flood of traffic to your site? Is your new social coupon site launching on TechCrunch? Did your Heroku-backed iOS app hit the top 10 in the App Store? You can simply and quickly scale up the database (along with the dynos) for a few days to handle the traffic, then scale back when ready.

- Need to test an important piece of code with a large production dataset? Create a Forked copy of the data and point your new code at the Fork.

- Worried about backups of your data? Rest easy. Heroku Postgres has you covered in multiple ways.

Swapping-out the database server

If your database server needs to be replaced, for whatever reason, using a Follower is the preferred method. Heroku uses the term 'Follower' to mean read-only replica. A Follower will stay in-sync with the main database until it is manually stopped, or 'un-followed'.

Suppose you are running a Fugu database, but have noticed recently that performance is erratic. It could be your application, but it could also be an underlying issue with the database server itself. Database servers are backed by disk volumes that can be problematic. Why not easily eliminate any server issues before digging into your application's code?

You can quickly add a Follower with this command -

```
$ heroku addons:add heroku-postgresql:fugu --follow COLOR_URL
```

The `COLOR_URL` above can be found via the the `heroku pg:info` command, and will be a different color value for each database server.

Once the above command is run, a new Follower will spin up and be available shortly, depending on the size of the dataset. You can check the status via the `heroku pg:wait` command. It's also worth noting that large datasets (10 GB or more) may show as ready, but indexes will likely take longer to fully rebuild.

Stop for a second and think about this. A fully-ready, in-sync read-only replica available in minutes with one command. Those of you who have configured databases servers by hand will truly appreciate how powerful this is.

When the new Follower is ready, make note of its `COLOR` value. Next, follow these steps to swap servers:

1. Place the application in maintenance mode.
2. Be sure all processes are scaled to zero.
3. Un-follow the new Follower, which will make it stand-alone and write-able.
4. Promote the new database to be the main database.
5. Take the application out of maintenance mode.
6. Remove the older database.

Step two is worth noting. It's important to turn off any process which could modify the database while the application is in maintenance mode. I find it's generally best to scale all processes to zero. You can see the full output of this process here:

```
$ heroku pg:info
=== HEROKU_POSTGRESQL_VIOLET_URL (DATABASE_URL)
Plan:        Crane
Status:      available
Data Size:   6.5 MB
Tables:      2
PG Version:  9.2.4
Connections: 6
Fork/Follow: Available
Created:     2013-05-15 02:24 UTC
Region:      eu-west-1
Followers:   HEROKU_POSTGRESQL_WHITE
Maintenance: not required

=== HEROKU_POSTGRESQL_WHITE_URL
Plan:        Crane
Status:      available
Data Size:   6.5 MB
Tables:      2
PG Version:  9.2.4
Fork/Follow: Unavailable on followers
Created:     2013-05-15 02:44 UTC
Region:      eu-west-1
Following:   HEROKU_POSTGRESQL_VIOLET
Behind By:   0 commits
Maintenance: not required
```

```
$ heroku maintenance:on
Enabling maintenance mode for debugging... done

$ heroku ps:scale worker=0
Scaling worker dynos... done, now running 0

$ heroku pg:unfollow HEROKU_POSTGRESQL_WHITE_URL
!    HEROKU_POSTGRESQL_WHITE_URL will become writable and no longer
!    follow HEROKU_POSTGRESQL_VIOLET. This cannot be undone.
!    WARNING: Destructive Action
!    This command will affect the app: csrbuddy-eu
!    To proceed, type "debugging" or …

> debugging
Unfollowing HEROKU_POSTGRESQL_WHITE_URL... done

$ heroku pg:promote HEROKU_POSTGRESQL_WHITE_URL
Promoting HEROKU_POSTGRESQL_WHITE_URL to DATABASE_URL... done

$ heroku maintenance:off
Disabling maintenance mode for debugging… done

$ heroku addons:remove HEROKU_POSTGRESQL_VIOLET

!    WARNING: Destructive Action
!    This command will affect the app: debugging
!    To proceed, type "debugging" or …
> debugging
Removing HEROKU_POSTGRESQL_VIOLET on debugging… done, v26 ($50/mo)
```

With the above steps, it's easy to swap-out the older database server. It's encouraged, and it's even the recommended way to upgrade versions when Heroku releases a new point-version of Postgres.

As you will see next, this swap-out process is the basis for other database operations on Heroku.

Scale up or down with a Follower

Taking the above Follower process one step further, imagine you are needing to scale up (or down) based on changing traffic. You can Follow an existing database with a different database size. You can even migrate from a Crane - the smallest production database, to a Mecha - the largest production database, via the Follower command. Migrating from a Mecha down to a Crane is also possible, or to any size in between.

This feature is so powerful that I want to stop and make note of it. I'm not aware of any other database service that does this as simply and powerfully as Heroku. There have been numerous times where customers have contacted support asking for help dealing with a large surge in traffic. Using a Follower to upgrade has saved many applications from being overwhelmed by traffic.

And the best part of all this database swapping? You will only pay for the time-used for each server. Even the larger servers are only a few dollars per day, and even less per-hour.

Using a Follower as a read slave

The most common use for a Follower is as a read slave in read-heavy applications. A great example of this is Heroku's internal billing system. At a basic level, the billing system continuously tracks events in a Postgres database. Heroku needs access to billing data in the form of reports, which are read-heavy. Running read-heavy operations against a write-heavy database is not good for performance, so reports are run against a Follower, which is effectively a read-slave. This enables Heroku to build near real-time reports while decreasing load on the main database used to track billing events.

Some applications are so read-heavy that it makes sense to use more than one Follower as a read-slave. This is known as sharding, and is one way to scale an application horizontally. One common example is using a Follower (or shard) for a pre-defined group in your application's data. For instance, it may make sense to split your customer data into two groups by last name, A through M and N through Z. You could then add two Followers to your application, and customer specific read operations would be sent to one of two Followers, depending on the customer's last name. To take advantage of a Follower, support will need to be added via code at the application level. If your application is built using Rails, you can use the Octopus library to enable ActiveRecord support for sharding via Followers.

In some cases, it can even be beneficial to use different sized Followers for read slaves. The main application database might be an Ika, but have several smaller Crane Followers to handle sharded read operations.

Forking a database

Creating a Fork is a feature unique to Heroku Postgres. A Fork is a point-in-time copy of a database. Forks are useful for testing, staging, and other situations where you need a copy of data but do not wish to modify the main database.

A common use of a Fork is a staging application. Perhaps you have a new set of features you'd like to test against production data, but you don't want to modify the production environment. You can Fork your production database to your staging application, as long as your Heroku account has access to both. This saves time, as the other way to do this, copying data from production to staging, would involve restoring a pgbackup from one application to another, something that is time-consuming for large datasets. With a Fork, you can do this with one command.

```
$ heroku pg:info -a debugging-staging
debugging-staging has no heroku-postgresql databases.
```

Above, you can see the `debugging-staging` application does not have a database, but we will create one by Forking another application's main database.

```
$ heroku pg:info -a debugging
=== HEROKU_POSTGRESQL_CHARCOAL_URL (DATABASE_URL)
...
```

We'll need only the `COLOR` name of the existing database for the Fork command.

```
$ heroku addons:add heroku-postgresql:crane --fork \
       debugging::HEROKU_POSTGRESQL_CHARCOAL_URL \
       -a debugging-staging
Adding heroku-postgresql:crane on debugging-staging... done, v3 ($50/mo)
Attached as HEROKU_POSTGRESQL_AMBER_URL
Database will become available after it completes forking
Use `heroku pg:wait` to track status.
Use `heroku addons:docs heroku-postgresql:crane` to view documentation.
```

The above Fork command creates a new database on the `debugging-staging` application that is a Fork of the `debugging` application's main database. After waiting for the Fork to come online, via the `heroku pg:wait` command, we can see the new database.

```
$ heroku pg:info -a debugging-staging
=== HEROKU_POSTGRESQL_AMBER_URL
Plan:        Crane
Status:      available
Data Size:   6.5 MB
Tables:      2
PG Version:  9.2.4
Connections: 3
Fork/Follow: Available
Created:     2013-05-16 14:02 UTC
Forked From: Database on ec2-54-228-194-85.eu-west-1.compute.amazonaws.com ...
Maintenance: not required
```

The new Fork is now ready for use by the `debugging-staging` application. You might have noticed something interesting in the above commands. The `debugging` application is running the Heroku EU Region, along with its main database, and the Fork was created in the Heroku US Region. With one command, we created a copy of the data in another geographic location. This is a very powerful feature.

Backups

Backups are another area where Heroku Postgres simplifies database maintenance. Heroku uses Continuous Protection to always protect your data. Simply put, Continuous Protection is the process of archiving Postgres WAL files to Amazon S3 for long-term storage. WAL is an acronym for for Write-Ahead Logging. You can see the source code for the Continuous Protection process here. The WAL file archiving happens every 60 seconds, so in a worst-case scenario, where Heroku would need to rebuild your database from the archived WAL files, you might lose 60 seconds of data (in my experience it's often much less than 60 seconds).

Continuous Protection is a great first-level backup. If you want another level of backups for your data, take a look at the `pgbackups:auto` add-on. This add-on will automatically backup your data in `pg_dump` format and store it on S3, where you can view the files and even download them for storage elsewhere.

Here's a sample of what the `heroku pgbackups` command does:

```
$ heroku pgbackups -a debugging
ID    Backup Time          Status                        Size   Database
----  -------------------  ---------------------------  -----  ------------
a080  2013/05/13 04:56.20  Finished @ 2013/05/13 ...    5.7KB  DATABASE_URL
a081  2013/05/14 04:58.36  Finished @ 2013/05/14 ...    5.7KB  DATABASE_URL
a082  2013/05/15 04:58.56  Finished @ 2013/05/15 ...    5.7KB  DATABASE_URL
a083  2013/05/16 05:44.54  Finished @ 2013/05/16 ...    5.7KB  DATABASE_URL
```

Each of the `a08X` entries in the above table are an auto-backup. You can download a backup via a URL:

```
$ heroku pgbackups:url a083 -a debugging
"https://s3.amazonaws.com/hkpgbackups/appxxxx@heroku.com/a083.dump?
AWSAccessKeyId=xxxx&Expires=1368714467&Signature=N8Vr0o3I5rdMVgoPILmZt3O%2FrjA%3D"
```

Note the above URL is public, but it will timeout and be unavailable after a few minutes. Using the above process, you could build a simple script to perform a daily download and store the data offsite.

PGExtras plug-in

The Heroku Postgres team is always improving the service and will often release utilities that make using the service easier. One of these utilities is the pg-extras plugin.

There are a variety of useful commands in pg-extras, but one of my favorites for debugging an application's performance is the `pg:cache_hit` command.

```
$ heroku pg:cache_hit -a debugging
       name      |         ratio
 ----------------+------------------------
  index hit rate | 0.99006561935013432170
  cache hit rate | 0.99768566842236281761
  (2 rows)
```

You can see the index and cache hit rates are 99% or above. In general, anything below 99% means the database server is working too hard and probably needs either better indexes or an actual server upgrade for more cache.

Summary

As you can see, Heroku Postgres is a powerful service that offers flexibility with your application's database not previously possible. With Fork and Follow new development workflows are possible. Be sure to give the Fork and Follow commands a try. There's no reason not to, and you will only pay for time-used.

CHAPTER 6

Using New Relic Effectively

New Relic is one of the most valuable and helpful tools I know of. If you're not aware of New Relic, or how it can help your application, I would suggest starting at NewRelic.com. At a basic level, New Relic provides visibility and metrics for how your application is performing. New Relic is so helpful that without it I feel blind when trying to asses the overall health or investigate a performance issue for an application.

New Relic is available as an add-on for nearly all Heroku applications. Ruby on Rails was the first framework supported, but now Python, Node.js, and Java applications are supported. Unless you have no need for visibility and metrics into how your application is performing, I recommend every application make use of at least the free New Relic add-on.

Installing New Relic is easy. First you need to enable the add-on:

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

Once enabled, you will need to configure your application to make use of New Relic. There are instructions in this Dev Center article. When you have made the necessary configuration changes to your application and deployed, New Relic will immediately begin gathering data as your application runs.

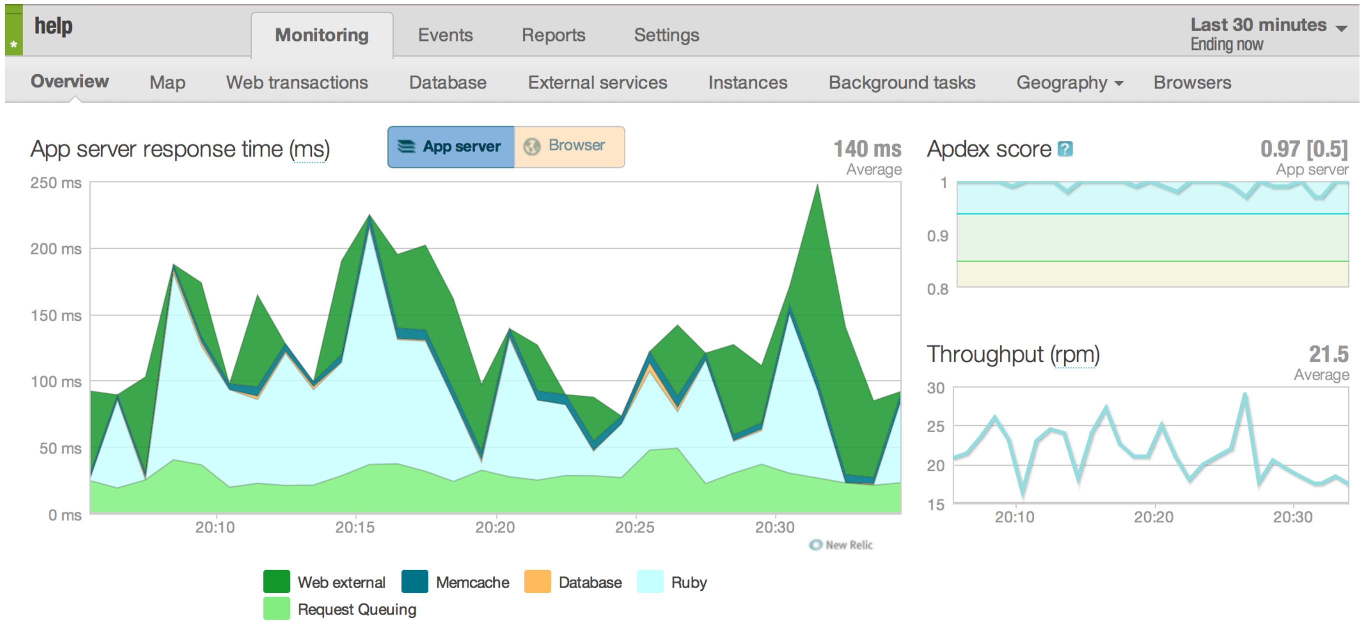
Your application's communication with New Relic is done asynchronously, which means you do not need to be worried about New Relic affecting performance. This is another reason I recommend every application use New Relic.

Overview Page

Once enabled and gathering data, you can open New Relic for an application with this command:

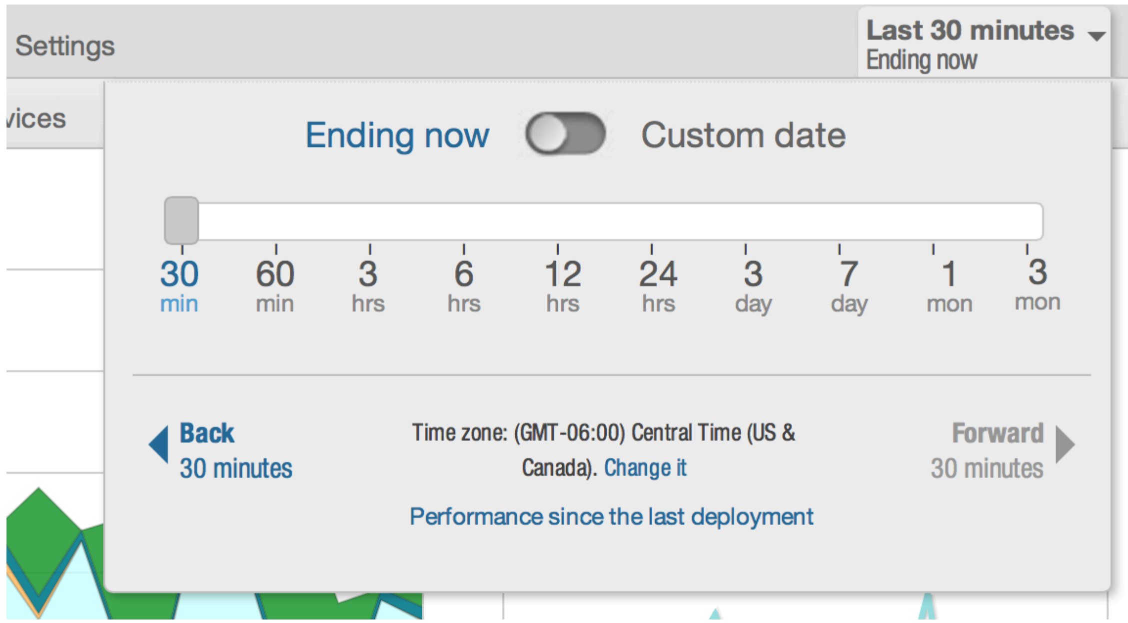
```
$ heroku addons:open newrelic
```

When first opened, New Relic will show an overview of how an application is performing. Here's an example:



The first thing you will notice is the large chart titled 'App server response time.' This chart shows an average of response times over the pre-defined timeframe. To see the timeframe, look to the upper right for a message such as 'Last 30 minutes Ending Now'. This indicates the timeframe New Relic is currently displaying.

It's worth noting the timeframe can lead to some confusion when trying to synchronize past events, such as log timestamps, with chart data. I like to set the timezone to either the timezone I am currently in or UTC. The Heroku log stream uses UTC for timestamps. With this in mind, here's how to adjust the timeframe:



Note the ‘Change it’ link at the bottom center.

The next chart is the 'Apdex score' in the upper right. This is a measurement of how well the application is performing for end users. You can hover over the ? next to the title of the graph for a more detailed explanation, but in general an Apdex score of 0.9 or above means the application is performing well.

The final overview graph is the Throughput measurement. New Relic uses RPMs, or Requests Per Minute, as its measurement of throughput. This is different than requests per second, another throughput measurement commonly used in load testing.

How else can New Relic be helpful? Here are a few examples:

- Comparing performance before and after a code deploy
- Tracking and notification of application exceptions
- Monitoring database throughput and performance
- Measuring, reporting, and tracing slow web or database transactions
- Measuring, reporting, and tracing slow database transactions
- Identifying traffic patterns over time
- Measuring third party API performance

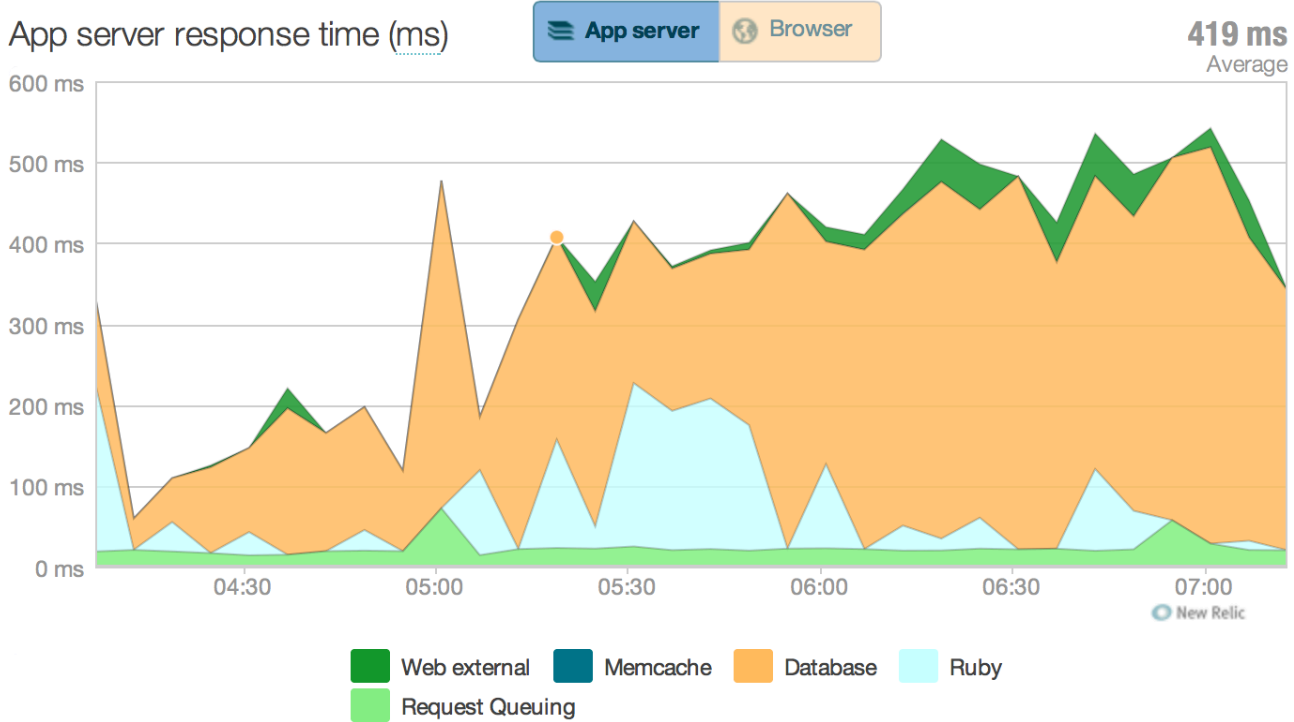
Example

I'm going to highlight 'Measuring, reporting, and tracing slow web transactions', as it's one of the most valuable features of New Relic.

Many times, customers will come to Heroku Support saying "my app is running slow, and nothing changed." The very first thing I do in this situation is look at New Relic. And if the application does not have New Relic installed, I'll ask the customer to install at least the free add-on before going any further. Also, it's worth noting that 'nothing has changed' is incorrect. There are many moving parts in even a simple application. New Relic can help find what changed.

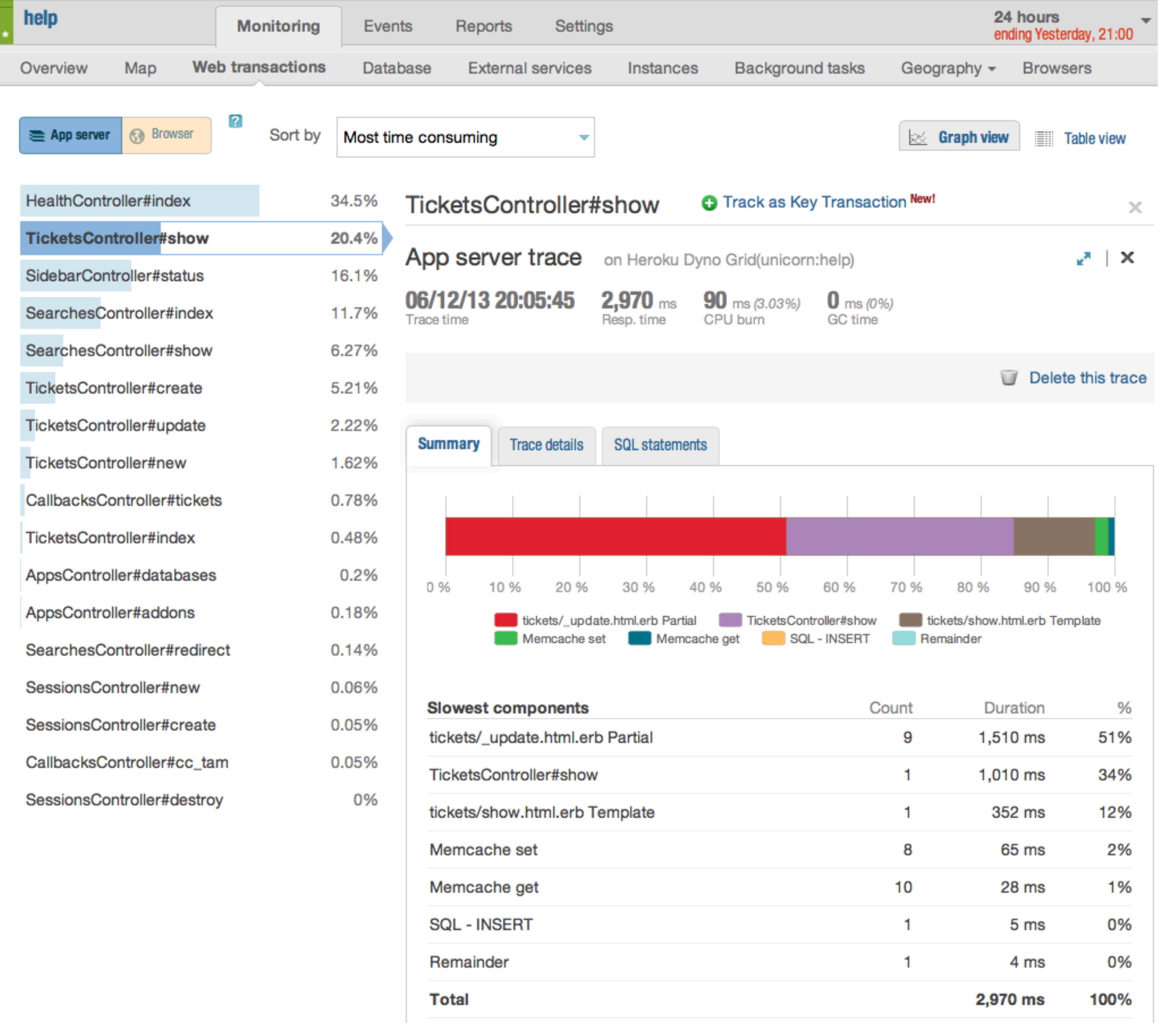
I usually start by setting the timeframe to 12 or 24 hours. The default timeframe of 30 minutes is not really helpful when investigating this type of issue.

Once the timeframe is set, I start with the main 'App server response time' chart. If the application is running slow, it will usually show here as spikes in response time. It's often possible to determine the cause of slowness just by looking over the main chart. For instance, if there is a noticeable increase in database time, this can indicate the database is slowing down the application. Other common problems include slow external services (usually a third party API), or even a slow caching layer such as Memcache or Redis.



Notice in the above chart, at 05:00, that database times spiked and caused the application to briefly slow down. It's also clear that as time progresses, this application is spending more and more time in the database. In the above chart, the database times actually see an increase of 400% over the course of two hours, from ~ 100ms to ~ 400ms. The application is still running fast, but if the 400% increase was from 1 second to 4 seconds, I would be concerned. This would be an indication the database server is under-powered or not indexed correctly.

If nothing stands out on the main chart, I'll look at the 'Web transactions' section next, under the Monitoring tab. The default sort, 'Most time consuming', is often enough to show which action(s) are problematic. Next, I'll click on the top action listed to get more detail. From here, it's possible to get even more details on the slow action, such as recent slow web transaction traces. Here's an example of a slow web transaction trace:



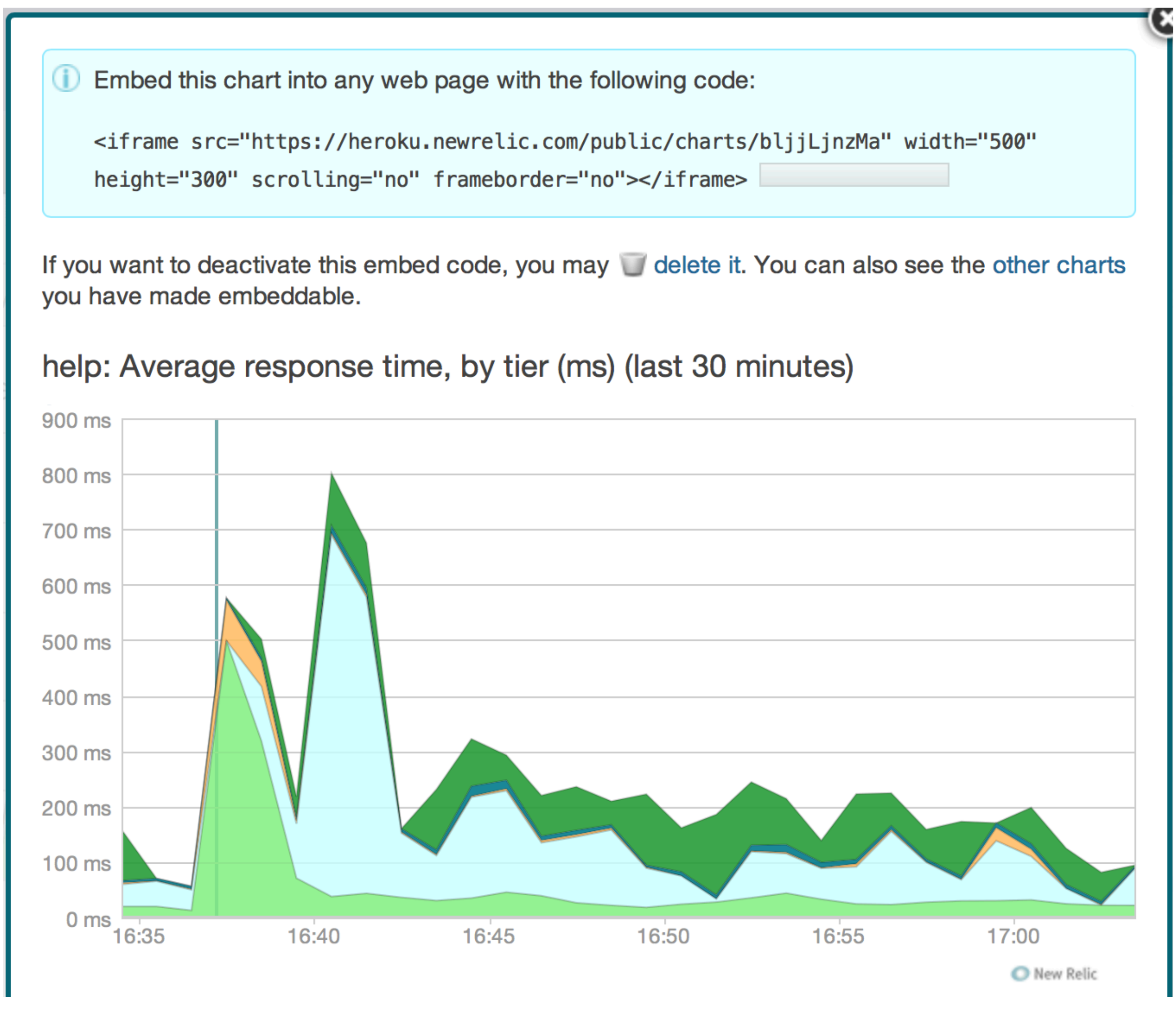
You can see that `tickets/_update.html.erb` is causing the transaction to run slowly, taking 1.5 seconds in this particular trace. Now we know at least one area that needs to be investigated and fixed.

At this point it's worth noting that some of New Relic's more advanced and helpful features require the paid version of the add-on. My suggestion, if you are running a true production application and need the performance analysis, is to pay for the New Relic add-on. It's expensive, but worth it. If cost is an issue, then I suggest upgrading for a few days while you investigate an issue, then downgrading to

the free plan when finished. You'll only pay for time used, and this is part of the flexibility Heroku add-ons provide.

Embedding Charts

New Relic has a nice built-in feature that allows you to create custom dashboards from charts or tables found throughout the site. These dashboards are handy, but sharing them with someone else requires the person to have a New Relic account. This is not always possible. However, it's still easy to share charts via the 'embed' link. Simply hover over a chart and look for an 'Embed' link in the lower right. This will pop up a screen that has some HTML <iframe> code. You can copy this code, or to share just the one chart, simply copy the the source URL and share. Here's an example:



This feature is very handy when you want to share the current RPM chart of an application, but don't want to grant access to all New Relic data. I make heavy use of this feature while working with customers over an extended period of time, as a simple way to keep an eye on important application metrics. I’ve also seen applications with a custom, admin-only status page, where select New Relic graphs are embedded in a page that can be easily shared with application owners.

Summary

Hopefully you see that New Relic is an invaluable tool for monitoring and measuring your application’s performance. When creating a new application, it's one of the first add-ons I implement. New Relic may seem complex and unnecessary at first glance, but once you've used it for a bit, you will keep coming back. The breadth and depth of what it measures and reports is almost un-matched.

CHAPTER 7

Customizing Heroku

The Heroku service has many conventions. An example of this is using `git push` for deploys. When you deploy code to an application via `git push heroku master`, the process of creating a release and starting the application is known as compiling the slug. Cedar is the default Heroku stack, however Cedar does not have a default language. Instead, when you deploy an application, scripts are run to determine the correct language or framework, and the application is then complied into a slug using a buildpack and your application's code.

It's the buildpack that defines how the application will run and what resources are available. Heroku provides a handful of common buildpacks that you can use, such as the Ruby or Node.js buildpack.

The interesting thing about buildpacks is that they are open source and customizable. Creating or modifying a custom buildpack is how you are able to customize Heroku to fit your application's needs. Custom buildpacks make it possible to define and run nearly any type of language, framework, or even single binary programs on Heroku. Remember, Heroku is just Unix. Buildpacks have their own simple API and can even be chained together.

But so what? You may already know about custom buildpacks. Perhaps you think custom buildpacks are a nice idea, but you're not sure how they can be used practically. Let's look at a great example of how a custom Ruby buildpack can save you both time and money.

Example - Hacking the Rails Asset Pipeline

I'm not sure how else to say it, but the Rails Asset Pipeline is cumbersome and frustrating. I agree that compiled assets are necessary, but the default implementation of the Rails Asset Pipeline is confusing. Rails 4 introduces a few changes to the Asset Pipeline that make it better, but with Rails 3.x the Asset Pipeline remains confusing, and this example focuses on Rails 3.x applications. The options for when to compile assets are minimal, and with Heroku most developers choose to compile assets during a deploy, rather than locally or during runtime. This is option is detected by the Ruby buildpack and handled by running the `rake assets:precompile` task near the end of the slug compilation process.

You might have noticed output like the following during a deploy:

```
-----> Preparing app for Rails asset pipeline
       Running: rake assets:precompile
       Asset precompilation completed (35.29s)
```

35 seconds is actually fast for asset compilation. The above time is the actual output for a deploy to the CSRBuddy.com application, and the number of assets in-use is minimal. As an application adds more and more assets, compile time will get longer and longer. It's not uncommon for assets to take several minutes to compile, and I've personally seen assets:precompile take ten minutes or longer.

It's worth noting that the slug compile process is limited to 15 minutes, so if asset compilation is taking too long, it's possible it will cause a deploy to fail.

If you are deploying a simple change to your application, and none of the assets have changed, this re-compile is a waste of time. And if you deploy frequently, as many developers do with Heroku, the compilation time begins to add up.

Fortunately, there is a way around this, using a custom Ruby buildpack. And it just so happens that somebody in the Heroku community has done the work for us.

Before diving into the code details, it's worth noting that buildpacks are given a cache directory during the compile process - https://devcenter.heroku.com/articles/buildpack-api#caching. It's this cache directory that will help us skip asset compilation.

Take a moment to review this pull request to the Heroku Ruby buildpack - https://github.com/heroku/heroku-buildpack-ruby/pull/96. The pull request is simple but effective. The key method is here - https://github.com/heroku/heroku-buildpack-ruby/pull/96/files#L2R98 - and I've listed the code below:

```
# Have the assets changed since we last pre-compiled them?
def precompiled_assets_are_cached?
	uncompiled_cache_directories.all? do |directory|
		run("diff #{directory} #{cache_base + directory} \
		--recursive").split("\n").length.zero?
	end
end
```

See how this buildpack uses the Unix `diff` command to check for changes to the assets? If there are changes to any assets, `rake assets:precompile` will run and store the compiled assets in the buildpack's cache directory. If there are no changes detected, the cached copy of the assets is used, thus saving time.

If you have a Rails 3.x application on Heroku and want to try this optimized buildpack, simply add this config variable:

```
$ heroku config:add \
BUILDPACK_URL=https://github.com/nthj/heroku-buildpack-ruby.git#precompile-optimizations
```

Once ready, deploy your application. You should notice the new buildpack being used as part of the first output of the slug compile process:

```
-----> Fetching custom git buildpack... done
```

Keep in mind that the first time you deploy with this custom buildpack, the `rake assets:precompile` task will run normally, but the results will be cached. Once finished, go ahead and make a minor change to the application, one that does not affect the assets. Then deploy again, and watch for this:

```
-----> Preparing app for Rails asset pipeline
       Assets already compiled, loading from cache
```

With the CSRBuddy.com application, deploys now run approximately 35 seconds faster.

In my opinion, this asset pipeline optimization perfectly illustrates the power and flexibility of Heroku buildpacks.

But what else can buildpacks do? Here are few buildpacks I find interesting and know to be helpful:

This buildpack allows you to chain multiple buildpacks together to combine functionality. Again, this is called 'composability'.

Want to manage your Postgres connections at the dyno level? You can do so via the pgbouncer buildpack.

You can run Nginx on Heroku via this buildpack.

Finally, there is an official list of third party buildpacks here.

Anvil

There is yet another way Heroku can be customized. Suppose your application needs additional binaries to function correctly. This could be any binary, but to keep this example simple let's say your application needs the `wget` binary for downloading files.

When building a binary to run within your application, it's important to keep in mind the underlying architecture used by Heroku. All Heroku dynos run on 64 bit Linux, so you will need to build on a similar system. Of course similar systems are not exactly the same as Heroku, so if you build a binary locally in OS X, there is no guarantee it will function correctly on Heroku.

The way around this is to build the necessary binary via Anvil, a build-as-a-service running on Heroku. Using Anvil, it's possible to fetch the source package for `wget`, then compile and build the binary on Heroku. The final result is a compressed package containing the `wget` binary which will run on Heroku.

Anvil provides several options for building, including using a local directory, a public Git repository, a custom buildpack, and even a shell script via URL. We'll use the last option for this example, providing a shell script via a gist. The shell script starts with the assumption the `wget` source package

has been extracted and its contents are available in the current directory. With that in mind, here's the shell script to compile `wget` with Anvil:

```
#!/bin/sh
root=$(pwd)
mv wget-* /tmp/wget
cd /tmp/wget
./configure --without-ssl
make
mv src/wget $root/
```

Notice at the end the `wget` binary is moved to the `$root` directory. This step is essential, otherwise the build result will not be available for use later. Let's look at a full example:

First, we'll verify that `wget` is not present on a new dyno.

```
$ heroku run bash
Running `bash` attached to terminal... up, run.5543
~ $ wget
bash: wget: command not found
```

Now, we'll use the `anvil` command to build `wget`, providing the full URL of the `wget` source package along with the build script.

```
$ anvil build http://ftp.gnu.org/gnu/wget/wget-1.14.tar.gz \
      -b https://gist.github.com/pdsphil/4c604dd53d1b9df09108/raw/build-wget.sh
Launching build process... done
Preparing app for compilation... done
Fetching buildpack... done
Detecting buildpack... done, Custom
Fetching cache... empty
Compiling app...
configure: configuring for GNU Wget 1.14
...
Putting cache... done
Creating slug... done
Uploading slug... done
Success, slug is https://api.anvilworks.org/slugs/5e30cc21-083c-4b49-8aed-db7ab2c3436c.tgz
```

When you run `anvil build`, the entire build output is shown, but I've omitted it from the example above. Notice the end result is a slug URL. Let's test the resulting slug via our `heroku run bash` dyno.

```
~ $ curl https://api.anvilworks.org/slugs/5e30cc21-083c-4b49-8aed-db7ab2c3436c.tgz | tar xzv
~ $ ./wget --version
GNU Wget 1.14 built on linux-gnu.
```

With the above in mind, it's easy to see how a custom binary can be configured, built, and included within your application. Extending this idea a step further, a custom buildpack could easily include calls to download pre-built binaries from Anvil, making the binaries available to the application.

Anvil has more advanced commands, including the ability to compile an application slug, and deploy the slug to various applications, bypassing the normal slug compile process. This creates some interesting workflow possibilities, but I'll leave those to you. Have a look at the Anvil project on GitHub for more examples.

Summary

Custom buildpacks are a powerful feature that allow you to customize Heroku to fit your needs. And with buildpack composability, it's possible to quickly build custom functionality using freely available buildpacks that others in the community have released. Add to this the ability to remote build binaries and buildpacks via Anvil, and it's easy to see how quickly Heroku can be customized. Customers are always surprising Heroku with new and innovative buildpacks. If you build something clever or useful, let us know!

CONCLUSION

My goal in writing this guide was to provide clear examples and helpful tips for developers running applications on Heroku. Hopefully, through the tips, debugging examples, and scenario walk-throughs, you now have a better understanding of how to run and support production applications on Heroku.

I've seen some amazing applications built on Heroku. If this guide helps your team, project, or application in some way, be sure to let me know! I love hearing stories about creative and unique ways the Heroku platform is being used.

It would be impossible to cover every scenario or possible issue in a single guide, so if you have questions or would like to see another topic explained, feel free to click the link at the bottom of this page and provide feedback. I do my best to respond to all emails, though it may take some time.

Thanks!

AFTERWARD

Interacting with Heroku Support

I spent the first two years working at Heroku as a member of the Support team. It's a small team (eight engineers as I write this), but it supports well over a million applications.

Your interaction with Heroku Support, if any, has probably happened via the ticketing system at https://help.heroku.com. I've probably answered at least one of your tickets if you've created one in the past two years. Hopefully I did a good job. The team does its best to be helpful, but all of us are human and make mistakes, so please keep that in mind the next time you interact with us.

Below are a few tips for interacting with the Heroku Support team. None of these are official, and some will change over time. Think of the following tips as something I would tell you were we discussing Heroku Support over drinks or a meal:

Create a ticket using the Heroku account that owns or is a collaborator on the application in question.

Support uses the account to look up further information about the application and attaches the data to the ticket. This helpful data will not be present if you submit a ticket via a personal email account with no access to the application.

If at all possible, grant access to look at the application's code.

The support team is not able to debug application specific issues, but having access to check configuration and initialization files is very helpful.

Include log data if applicable.

There's a reason the `heroku logs` command is the first chapter in this guide. Without the log data, it can be difficult or impossible to diagnose the issue. And the support team is often unable to look at older log data. The correct snippet of log data included with a ticket can help resolve the issue quickly.

Heroku's support hours are 6 AM to 6 PM Monday through Friday, Pacific time.

Support does its best to respond to your ticket within one business day. Yes, this means that if you submit a ticket on Saturday, you will probably not get a response until Monday. There are Premium support plans if you require a 24/7 SLA response.

If you have more than a handful of lines of code or log data to share, use a gist.

It's very difficult to track longer tickets when there are dozens or hundreds of lines of code or log data interspersed with dialogue. Also, please do not include passwords or credentials to databases or add-ons. If you do accidentally include a set of credentials, you'll likely be asked to change them.

Support issues are not handled via Twitter.

Twitter does not work well for complicated issues. At best, someone will respond to your question or issue and ask you to open a ticket.

If you are debugging a strange issue, be sure to include steps to reproduce the issue.

A simple `curl` command is often enough. In particular, the `curl -I` command is useful when debugging asset caching issues. If the issue can be reproduced via a series of steps in a web application, include those steps along with a test account if needed.

Get to know Markdown, if you don't already.

The ticketing system supports markdown, and it will help to properly format code or terminal output.

Prioritize your ticket correctly.

'Urgent' means your *production site is down*, not that you are unable to deploy to staging or you have a question about SSL. If you have a Premium support plan, using an Urgent status will trigger alerts and page an on-call engineer, including waking them up. I understand how frustrating it can be to run into an error with your application in the middle of the night or over the weekend, but please, please, please use 'Urgent' carefully.

Don't be surprised if you see several members of the support team answering your ticket.

The team discusses various issues and tickets throughout the day, and often hand off a ticket or jump in with answers when needed.