PART I
An Overview of Heroku

- **CHAPTER 1:** How Heroku Works
- **CHAPTER 2:** Architectural Considerations
- **CHAPTER 3:** Porting Your Applications to Heroku

1

How Heroku Works

WHAT'S IN THIS CHAPTER?

- How the stacks stack up
- Understanding dynos and the dyno manifold
- Understanding dyno isolation
- Exploring the process model
- Understanding erosion resistance
- Managing version control
- Understanding the slug compiler
- Routing HTTP requests

Heroku is a polyglot, cloud-based development and application-delivery platform. It helps developers focus on building apps by removing the need for servers, system administration, and stack maintenance. The Heroku platform is a multi-tenant architecture built on virtual machines in Amazon Elastic Compute Cloud (Amazon EC2). Application management and scaling is done either through a command-line interface or online and on mobile devices, wherever you happen to be, via the API. Apps built on Heroku benefit from Heroku's managed stack and self-healing architecture, ensuring that they do not require system administrators to manage the underlying platform in order to keep them secure and reliable. Automated failover, disaster recovery, and bit rot prevention are built in to the platform.

The Heroku platform offers capabilities for near-immediate deployment for both development and production apps. Heroku greatly assists agile development methodologies and allows for seamless continuous deployment. Configurable access security enables individual members of the development or scrum team to push changes. Overhead to set up and manage software development life-cycle support environments is no longer a chore, with Git's version control system built in, automated dependency management, and system rollback available with a

single command. Developers can leverage a readily available Postgres database, using either SQL or key-value data storage, or use their own external data store of choice. Add-ons for enhanced features such as monitoring and deploy hooks are readily available. This provides marked gains in developer productivity and enables an organization to both predict and reduce management costs.

The first part of this chapter discusses stacks, the deployment environment used to run the apps you develop. It then covers the dyno manifold, which is the hands-off process management environment, built on a cloud-computing platform. After outlining the dyno isolation process, which enables you to manage different process types in a rapidly scalable, fault-tolerant environment, you will learn how code is deployed to be run as a dyno, and how version control is enforced with Git. We will then take a look at the benefits that this architecture offers to prevent application erosion and to enable coding in an operations-free environment. Next, we'll discuss how version control is managed with code deployed to Heroku, and how this code is packaged and run on Heroku with the slug compiler. Finally, we will describe how HTTP requests are routed to your deployed Heroku app.

HOW THE STACKS STACK UP

Each stack is a base operating system, language execution environment, and application serving software. There are different stacks to support different operating environments provided, as detailed in the following table. The stack sits on top of a virtually isolated Linux-based machine in Amazon's managed data center. Stacks enable you to run your app in the cloud without having to consider the underlying infrastructure.

STACK	BASE TECHNOLOGY	MRI 1.8.6	REE 1.8.7	MRI 1.9.2	NODE .JS	CLOJURE	JAVA	PYTHON	SCALA
Argent Aspen	Debian Etch 4.0	X							
Badious Bamboo	Debian Lenny 5.0		X	X					
Celadon Cedar	Ubuntu 10.04		X	X	X	X	X	X	X

The only stack on which you can currently create new apps is Celadon Cedar. Cedar is a versatile polyglot stack that enables development with a supported, managed stack for apps coded with Ruby, Java, Python, Node.js, Scala, or Clojure. Cedar is flexible enough for you to architect and manage your own custom stacks with various languages, development frameworks, and libraries of your own choosing while still taking advantage of the managed infrastructure and deployment tools built

in to the platform. Leveraging the stacks provided enables hands-off operations without the need for patching, and includes the latest security updates and reliability improvements for the operating system and language run times without requiring user intervention.

Prior to Cedar, the default stack was Badious Bamboo, which supports Ruby 1.8 and 1.9 app development when running on a Debian 4 OS. Both MRI (Matz's Ruby Implementation) and REE (Ruby Enterprise Edition) were supported. Argent Aspen was the original stack created for the deployment of Ruby 1.8.x apps, both run on Debian. The largest difference between these two and Cedar, beyond the polyglot capabilities in Cedar, is that both Aspen and Bamboo provide a caching layer using Varnish, a reverse proxy and HTTP accelerator, which is no longer a recommended approach. You can no longer create apps on Aspen or Bamboo. Aspen is end of life, and Bamboo is expected to be very shortly.

The Heroku command line enables you to control your local code base and remote Heroku apps in a terminal running on Windows, Mac, or Linux. After installing the Heroku command-line tool, which is included with the Heroku Toolbelt (see Chapter 4, "Deploying Your Applications to Heroku"), you can see the currently available stacks using the command line:

```
$ heroku stack
  aspen-mri-1.8.6
  bamboo-mri-1.9.2
  bamboo-ree-1.8.7
* cedar (default)
```

The command-line tool runs within the context of the current directory of your command line (Windows or Mac Terminal, bash, etc.) and interacts with Heroku's API. In the preceding example, the asterisk beside the stack name indicates which stack the current app (within the context of the current directory) is running. You can use this to create an app that will run on the stack of your choice, as follows:

```
$ heroku create
Creating young-summer-9942... done, stack is cedar
http://young-summer-9942.herokuapp.com/ | git@heroku.com:young-summer-9942.git
```

The preceding example creates an app that will run on the default Cedar stack. Simply running `heroku create` without additional flags will provision an app run time on Heroku's current default stack. At the time of writing, the default stack is Cedar. Some stack features may have beta status, meaning they are still undergoing testing and improvement and should therefore be used with caution for production-grade apps, as interfaces (APIs and versions of stack components) may change. When running apps with beta components, developers may be required to make infrequent, minor changes to ensure that their apps are compatible with the latest interfaces provided. Apps can be upgraded to run on subsequent stacks, in order to take advantage of enhancements or improvements. An automated process for this migration is documented on the Heroku Dev Center (`https://devcenter.heroku.com/articles/cedar-migration`).

If you have automation scripts and tools, you may choose to include the `--stack` flag, because the default stack may change in the future, resulting in deployment to the incorrect stack after the change is made.

> **NOTE** *This book focuses on the current default stack, Cedar. All the examples and discussions use this stack.*

UNDERSTANDING DYNOS AND THE DYNO MANIFOLD

Stacks and the accompanying code will run on a *dyno*, which is a single process of any type running from your app on the Heroku platform. Each dyno can run a variety of different types of processes, including but not limited to the following:

- Web apps that serve pages
- Processes that serve API requests
- Time-based batch or background workers
- One-off programs

The *dyno manifold* can be thought of as a giant, distributed, fault-tolerant, horizontally scalable computer in the sky, capable of running a nearly infinite number of diverse programs in the cloud. It offers you complete and instantaneous control over your process formation, and it is completely hands-off and maintenance-free once your processes are up and running as dynos.

Dynos offer users a number of features and advantages:

- **Scaling elasticity:** Processes can take advantage of the inherent elasticity of a cloud-based infrastructure, enabling developers to run as many instances of the process as needed at any given time. Different copies of each process can be scaled up and down across one or more dynos to provide near instant and nearly unlimited scalability to adapt to demand. For instance, if you have a background process that takes a very long time to complete on a single dyno, it can be run on another dyno in parallel, creating a divide-and-conquer solution to complete the task at hand in significantly less time. A running web app that begins to slow under heavy demand (perhaps an app has "gone viral" or is experiencing load spikes) can be scaled in seconds to serve more requests at once.
- **Intelligent routing:** The dyno manifold also provides automated load balancing that does not require complex configuration. HTTP traffic from your hostname(s) is intelligently directed to the correct web dyno inside the dyno manifold. The `herokuapp.com` HTTP stack offers a direct routing path to your web dynos. This allows for advanced HTTP uses, such as chunked responses, long polling, and using an asynchronous web server to handle multiple responses from a single web process. Traffic management is handled by a load balancer that offers HTTP and SSL termination. From there, requests are directly passed to the routing mesh. This shorter path enables full support of HTTP 1.1 but without offering any implicit caching capabilities.
- **Process management automation:** Processes that are unresponsive or misbehaving are restarted automatically. Continuous monitoring ensures that this is done without user intervention, routing future requests to available and responsive dynos while the existing

dyno is restarted. Dynos typically are cycled once a day or when degradation or failure in the underlying hardware is detected and fully logged. Because of this ephemeral nature of a dyno, apps must be written in a manner that does not rely on a particular instance, in close alignment with best practices for horizontally scalable, stateless app design. (For more information, see Chapter 2, "Architectural Considerations.")

- **Distribution and redundancy:** Because your process can run on multiple dynos, failover and redundancy are built in. When multiple dynos are running in parallel, processes are automatically distributed onto different physical hardware, providing protection from a single point of failover on a particular server. Dynos are run in multiple Amazon data centers, providing geographical redundancy and automated backup.
- **Isolation:** Each process is run completely separately from the others. Individual processes run in a hardened container that prevents other apps from using the same memory or filesystems, which enforces fair allocation of processor resources on the server. In a multi-tenant environment, this behavior is very important to ensure security and reliability when running multiple apps on the same infrastructure.

Each dyno is allocated 512MB of memory, which is isolated from other dynos and processes to ensure that any memory leaks that occur do not affect other apps running on the same hardware. It also provides protection against violations of memory safety introduced by events such as buffer overflows, thereby minimizing security risks and vulnerabilities for apps run on the platform.

Exceeding the 512MB of memory on a dyno neither prevents an app from executing nor kills it. Instead, additional memory required over the 512MB allocation will go in to swap. This results in sub-optimal performance of apps on the system, as I/O requests are often many orders of magnitude slower than random memory access. If memory usage exceeds three times the limit ($3 \times 512\text{MB} = 1.5\text{GB}$), the app is automatically restarted. You can also restart your entire app manually at any time, by entering the following:

```
$ heroku ps:restart
Restarting processes... done
```

You can also restart individual dynos manually by using the preceding command followed by the name of the process:

```
$ heroku ps:restart worker.12
Restarting worker.12 process... done
```

This will trigger a `SIGTERM` signal to all processes, allowing your app, running processes, and the underlying software stack to optionally catch the signal and do what it needs to do to shut down gracefully.

Apps that have only one web dyno will idle out after an hour of inactivity. When idling, it takes a few seconds for the dyno to wake back up and start processing requests, causing a short delay in responsiveness to your app. When your app has more than one dyno, subsequent dynos and worker dynos are never idled.

UNDERSTANDING DYNO ISOLATION

As stated earlier, each process in Heroku runs in isolation from one another, despite the fact that a number of different processes may be running on a single virtual instance. Heroku ensures that each dyno has its own independent filesystem and that memory and system resources are evenly distributed. Isolation also ensures that overloaded apps cannot compromise other apps running on the same virtual machine or server, and even isolates processes from the underlying operating system itself. Process isolation is achieved using LXC (Linux Containers) and `chroot`.

LXC is used to ensure that each dyno runs within its own container on an Amazon EC2 virtual machine. Therefore, in essence, you can think of Heroku as a virtual environment running in a virtual machine. LXC is a lightweight virtual system mechanism that its own website refers to as "chroot on steroids." LXC can emulate thousands of systems on a single server using Linux's built-in control groups (cgroups) functionality, which provides process containers. Dynos running within an LXC cannot see other processes running on other dynos because of process namespacing built in to the OS kernel.

Each dyno also has its own filesystem, which is kept completely separate from filesystems on other dynos using `chroot`. This filesystem includes the OS and stack used to execute your code, and the last deployed copy of the code itself. You can write files to this filesystem; however, because of the temporary nature of individual dynos, which are continuously brought up and down and cycled regularly, this filesystem is ephemeral and should not be used for permanent storage of any sort. Instead, for longer-term storage, you can store content to be used as assets in services such as Amazon Simple Storage Service (Amazon S3). The share-nothing nature of dynos also makes the local filesystem inappropriate for storing session information, as subsequent web requests could be routed by the load balancer to other dynos without the session information.

EXPLORING THE PROCESS MODEL

The Heroku process model is very similar to the UNIX process model, so if you are familiar with that you can consider them to work in the same way — that is, enabling diverse process types, not just simple web processes that wait for web requests. The process model also enables capabilities such as background processes that divide up the work and complete tasks in parallel on Heroku. However, because many dynos are running on a single virtual instance, the dyno manifold provides a distributed process manager.

Typically, when you start up a server, you need to do things such as start cron jobs and the web server. In traditional UNIX systems, you would add these programs to `init.d` to have them start automatically. For example, a Rails app will typically have a web server, such as Webrick, Unicorn, or Thin, as well as a library for queuing and asynchronously running worker processes, such as Resque, BackgroundJob, or ActiveMessaging. Different processes need to be run for different apps, and this information must be defined somewhere.

Therefore, using the Rails example, when you start up your web server, you want the processes defined in the following table to start automatically:

PROCESS TYPE	INVOCATION COMMAND
Web	`bundle exec rails server`
Worker	`bundle exec rake jobs:work`

Similarly, if you want that web server to run Python apps on the Django framework, the following table demonstrates the processes:

PROCESS TYPE	INVOCATION COMMAND
Web	`python manage.py runserver`
Worker	`celery –loglevel=INFO`

You can define which process invocation command you want your dyno manifold to execute on dyno creation using a file named `Procfile` that is placed in the root of your app. In the preceding example, you would set your `Procfile` as follows:

```
web: bundle exec rails server
worker: bundle exec rake jobs:work
```

Although this example uses the process type names `web` and `worker`, you could use any names you want to describe your process types. The only special process type name is `web` because this is the process type that receives HTTP traffic. The `Procfile` uses the following simple format for each line:

```
<process type>: <invocation command>
```

Note that the `Procfile` is optional; if it is not found when you push your code, the dyno manifold will detect the language automatically and launch the default web process. To see what processes are running, use `heroku ps`:

```
$ heroku ps
Process       State               Command
------------  ------------------  ------------------------------
web.1         up for 7m           bundle exec rails server mongrel -..
web.2         up for 2s           bundle exec rails server mongrel -..
worker.1      up for 7m           env QUEUE=* bundle exec rake resque:work
worker.2      up for 3s           env QUEUE=* bundle exec rake resque:work
worker.3      up for 2s           env QUEUE=* bundle exec rake resque:work
worker.4      up for 3s           env QUEUE=* bundle exec rake resque:work
```

You can see from the first column that two web dynos and four worker dynos are currently running. This command also returns the state of the process and which command was invoked (defined in the `Procfile`, if one were found). Note that when you scale up your apps, you can scale each process type independently, providing an additional level of granularity over virtual machines. For more information on scaling, see Chapter 7, “Managing, Monitoring, and Scaling Your Heroku Applications.”

You can schedule a process to run at a specified time of day or time interval (similar to `cron` in UNIX) using the Heroku Scheduler. See Chapter 6, "Working with Add-Ons," for more information.

Sometimes, you will need to run a one-off process such as initializing or migrating a database, or run a one-time script that sits somewhere in the committed code. You can run these one-off processes using the `heroku run` command. However, this is very different from running a process formation (like the previous example with web and worker processes) in a number of different ways.

When using `heroku run`, your local computer's terminal is attached to the dyno. This new dyno uses the naming convention `run.N` (instead of `web.N` or `worker.N`). Anything you type is interpreted as standard input (`STDIN`) on the dyno. Standard output (`STDOUT`) is typically pushed to the log; but when you are using `heroku run`, `STDOUT` is redirected to your terminal, unless you use the `heroku run:detached` command.

To terminate this session and kill the dyno, press Ctrl+C or use `heroku ps:stop <process name>`. Dynos are not restarted or cycled while using `heroku run`; and these dynos will never receive HTTP traffic like web dynos, as HTTP traffic is routed only to web dynos. (For more information, see the "Routing HTTP Requests" section.)

Using `heroku run` can also provide interactive console access to your app:

```
$ heroku run console
Running `console` attached to terminal... up, run.1
irb(main):001:0>
```

This is helpful for interactive run time or debug sessions against your app or to launch one-off jobs. Note that the session will remain active with an allocated dyno running until you run `exit` to terminate the session.

UNDERSTANDING EROSION RESISTANCE

One of the main benefits of using the Heroku platform is its built-in mechanisms to self-heal, often without the need for user intervention. When running your app on a server over time, entropy will almost certainly creep in at some point, resulting in crashes due to stack issues that have not been patched, log files filling the server's available disk space, or apps crashing or freezing, which requires killing and restarting the app or the server. This is commonly referred to as *bit rot* or *software erosion*.

The dyno manifold provides erosion resistance through a number of different techniques. First, dynos are typically cycled on a daily basis. In the background, the dyno manifold spawns a fresh dyno using the most recently pushed version of the code base and the latest version of the managed stack. Old dynos are brought down gracefully after they have finished their current work and replaced with this new dyno, with no perceived downtime to the user. This means that if any updates have been made to the underlying stack, they are automatically patched through this process, giving you the most stable and secure version of the stack to run on without the need to manually patch and restart your server. Minor revisions are used for these patches, so stack updates should not break your app.

Second, the dyno manifold also detects crashes that occur, either during startup or operation. If this happens, the dyno manifold will automatically restart the dyno. If it crashes again, another restart

is attempted after a ten-minute cooldown period. If you want to restart manually during this period, you can do so using the `heroku ps:restart` command.

If an underlying hardware failure is detected by Heroku's robust app-monitoring system, the dynos running on that system are automatically restarted on different hardware. This enables Heroku users to take a no-operations approach to running their apps. Expensive monitoring software is no longer required, as fault tolerance is built into the system. Heroku's team of engineers wear the pagers, so system administrators are not needed to "keep the lights on" or continually keep on top of patches for each element of the stack and install them. This minimizes app downtime and avoids the high operational costs associated with software erosion, enabling an organization using Heroku to focus on innovation, rather than ongoing monitoring and maintenance.

MANAGING VERSION CONTROL

Heroku has a built-in version control system, so you don't have to build and configure your own. Git is an open source, distributed, revision control and source control management system. It was originally designed and created by Linus Torvalds for maintaining the source code for the Linux kernel, after Torvalds was criticized for using a proprietary solution. Git creates a `.git` directory in your local code repository that contains all the versioning and historical information for your code base. This is a paradigm shift from traditional systems that rely on a central server, often remote, to manage revision tracking. Git is also very fast and scalable, making it ideal for use with projects of any size.

In addition to providing source control management, Git is the deployment mechanism used to push code to Heroku. Deployment is done via GitHub, an online code-hosting service using Git. Though GitHub has both free and paid accounts available, a paid account is not required to use this service with Heroku.

Deploying your code takes two commands — one to commit your code and another to push it to Heroku:

```
$ git commit -am "My latest change"
[master a3f713b] My latest change
 1 files changed, 1 insertions(+), 1 deletions(-)

$ git push heroku master
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 432 bytes, done.
Total 6 (delta 2), reused 0 (delta 0)

-----> Heroku receiving push
```

After you have run these commands successfully, your code will be deployed in production to all dynos running your app's processes. You can also use this process to deploy your app to different development environments. For instance, if your team's deployment process requires that you first test your changes in a staging or quality assurance (QA) environment, the `git remote` command enables you to manage multiple environments for an app without having to provision and maintain multiple servers, keep the stacks identical between environments, or write scripts for deploying to each environment.

Heroku's easy deployment and built-in code management explain why developers love using it. In traditional development environments, one must install a source control management system somewhere, maintain the repository, and keep backups and provide disaster recovery if something happens. Deploying code to multiple servers also requires a lot of additional work in terms of writing and maintaining deployment scripts that keep track of the ever-changing list of servers that must be deployed to. In addition, because the code repository is centralized, developers must have access to this server, either on-site at the office where it is firewalled or via a virtual private network (VPN), which can be expensive and complex to set up. This all results in more unnecessary operational overhead and headaches for the developers, when all they want to do is deploy their code.

> **NOTE** *For more on using Git and GitHub, see Chapter 4.*

UNDERSTANDING THE SLUG COMPILER

Before pushing your code to Heroku, you first must compile and package it. During this process, your code is sent to Heroku's slug compiler, which will get the latest copy of your code from the head (the most recent commit) of the master branch of your repository, install dependencies needed to run your app, compile your code (if needed), and deploy this "slug" to each of your dynos. The slug is stored on Heroku's servers; therefore, when scaling up, the slug is reused, making scaling very fast, with no need to recompile the code.

Consider the following example push of a Ruby app:

```
$ git push heroku master
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 483 bytes, done.
Total 6 (delta 4), reused 0 (delta 0)

-----> Heroku receiving push
-----> Ruby/Rails app detected
-----> Installing dependencies using Bundler version 1.1.2
       Running: bundle install --without development:test --path vendor/bundle
       --binstubs bin/ --deployment
       Using rake (0.8.7)
       [...]
       Using rails (3.1.1)
       Your bundle is complete! It was installed into ./vendor/bundle
       Cleaning up the bundler cache.
-----> Writing config/database.yml to read from DATABASE_URL
-----> Preparing app for Rails asset pipeline
       Running: rake assets:precompile
       (in /tmp/build_3p1i65rctat1c)
       /usr/local/bin/ruby /tmp/build_3p1i65rctat1c/vendor/bundle/ruby/1.9.1/bin/rake
       assets:precompile:nondigest RAILS_ENV=production RAILS_GROUPS=assets
       (in /tmp/build_3p1i65rctat1c)
```

```
-----> Rails plugin injection
       Injecting rails_log_stdout
       Injecting rails3_serve_static_assets
-----> Discovering process types
       Procfile declares types      -> (none)
       Default types for Ruby/Rails -> console, rake, web, worker
-----> Compiled slug size is 17.3MB
-----> Launching... done, v13
-----> Deploy hooks scheduled, check output in your logs
       http://freezing-day-9626.herokuapp.com deployed to Heroku
```

Note that during the slug compile process, the compiler first automatically detects the app type. In this case, a Ruby on Rails app is being deployed. Next, Bundler, the Ruby app dependency manager, installs your app's dependencies. Then, configuration variables (also known as *config vars*) are used to inject environment-specific information — for instance, the database URL and login info, which might be different in staging than production; see Chapter 5, "Managing Releases in Heroku" for more details. After this, process types are discovered. Default types that are created automatically appear here, as well as any process types declared in the `Procfile` (refer to the "Exploring the Process Model" section). Finally, the slug is compiled and this latest version of your app is launched on each of your dynos running on Heroku.

During compilation, the `.git` directory and any unnecessary files (for instance, with a Ruby app, anything in the `log` or `tmp` directory and any local `.gem` files) are not sent to the slug compiler. A `.slugignore` file in the root directory can also be used to indicate any other files or directories that should be ignored by the slug compiler. For instance, unit tests, design specs, and art originals (such as Photoshop files) are good candidates for the `.slugignore` file. Files used in the `.gitignore` file are also excluded from the slug compiler.

Each file or directory is specified on a single line in the `.slugignore` file, as follows:

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

This particular `.slugignore` file ensures that `.psd` and `.pdf` files are not included, as well as the contents of the `test` and `spec` directories.

During the compile process, the slug file size is reported. The maximum size for any slug is 200MB. However, if your slug is over 50MB, you should consider minimizing it. This can typically be done by using the `.slugignore` file to remove unnecessary files and/or moving your static assets (PDFs, images, etc.) to an external file storage service such as Amazon S3. Large slugs take the dyno manifold longer to deploy, resulting in slower scaling.

ROUTING HTTP REQUESTS

HTTP requests are automatically routed to your app's web dynos. These requests are sent to load balancers that provide SSL termination, and are then passed down the routing mesh to be intelligently distributed to the dyno with the least workload to serve the response. This load balancing provides optimized throughput, minimizing response time, as well as maximum reliability and redundancy when multiple dynos are available.

Load balancing is always handled automatically within Heroku. As shown in Figure 1-1, anytime you add or remove dynos from your app, Heroku will add or remove those dynos from the routing layer of the platform. Each request is routed to a randomly chosen dyno. Note that Heroku does not support sticky sessions or session affinity.

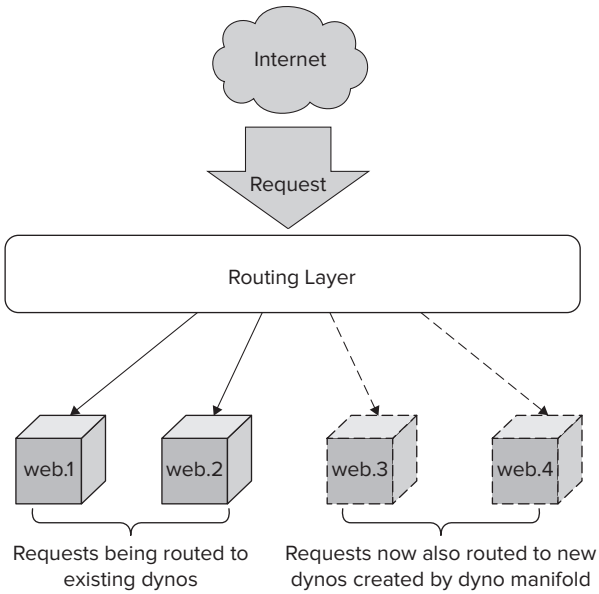

FIGURE 1-1

Apps that are deployed to the default Cedar stack offer a direct routing path to dynos through the `herokuapp.com` HTTP stack. Requests to `yourappname.herokuapp.com` use this stack, which fully supports HTTP 1.1 except for implicit caching. HTTP 1.1 allows for chunking, which transmits responses in a series of fragments, optimizing response time when large amounts of data are returned, as content can begin transmission before the entire content payload is dynamically generated.

Requests are made directly to the application server, not proxied through an HTTP server such as nginx. Therefore, response compression must be performed in your app; and because requests are being made via HTTP, WebSockets and incoming TCP connections outside of web ports are not natively supported. Third-party Heroku add-ons, such as Pusher (`https://addons.heroku.com/pusher`), enable you to add this functionality to Heroku apps.

> **NOTE** *See Chapter 6 for more information on how to leverage add-ons.*

Any requests that arrive at the Cedar stack must return a response data within 30 seconds; otherwise, the connection will time out and terminate. Once an initial response is sent, subsequent responses or requests must be made within 55 seconds to keep the connection alive. Long polling

is supported, allowing a request to be held if no information is immediately available. When information is available, or the connection subsequently times out, a response can be transmitted.

Multi-threaded apps can be utilized to enable apps to maintain multiple connections simultaneously and do asynchronous processing. For instance, Ruby web servers such as Goliath and Thin (with a suitable web framework such as Async Sinatra) can take advantage of multi-threading in a single process. You can also spawn and control your own threads if, for instance, you write a Java app or Node.js app built with the Express framework that uses this functionality.

The Aspen and Bamboo stacks provide built-in HTTP acceleration using Varnish. Cedar apps do not provide this, in order to support advanced HTTP features that Varnish does not support. Couchbase's Memcache add-on provides scalable, in-memory object caching, based on the popular memcached system. Using this add-on with `Rack::Cache` or `Radiant::Cache`, Ruby developers can speed up delivery by reducing database load, often the biggest bottleneck to an app's performance. (For more information, see the "Caching with Ruby" section of Chapter 11, "Developing with Ruby.") Content delivery networks (CDNs), such as Amazon S3 and CloudFront, can be leveraged to speed up page load times through distributed local delivery of static assets.

SUMMARY

This chapter provided a high-level overview of Heroku. We explored Heroku's stacks, including the current default, Celadon Cedar. You should also have an understanding of dynos and how they fit into the platform architecture via the dyno manifold, as well as what dyno isolation means to your app and how to leverage it. You should comprehend how versioning in Heroku works and how the platform ensures your app remains erosion resistant as the platform evolves over time. Digging deeper, you saw how app delivery is possible via app slugs and the slug compiler, which builds apps when they are ready to be fired off into dynos. You also gained insight into the unique HTTP routing that Heroku provides, regardless of wherever in the platform your app resides, or whichever dynos you may leverage. Heroku's routing will always funnel requests properly to ensure your app is serving requests with optimal performance. This emphasis on performance is also at the core of Chapter 2, which explores the architectural considerations you'll need to keep in mind as you design, build, and deploy apps on the Heroku platform.