

SAMPLE CHAPTER



JavaScript

Application Design

A Build First Approach

Nicolas Bevacqua

FOREWORD BY Addy Osmani



JavaScript Application Design
A Build First Approach

by Nicolas Bevacqua

Chapter 4

Copyright 2015 Manning Publications

brief contents

PART 1 BUILD PROCESSES 1

- 1 ■ Introduction to Build First 3
- 2 ■ Composing build tasks and flows 23
- 3 ■ Mastering environments and the development workflow 50
- 4 ■ Release, deployment, and monitoring 71

PART 2 MANAGING COMPLEXITY 97

- 5 ■ Embracing modularity and dependency management 99
- 6 ■ Understanding asynchronous flow control methods in JavaScript 131
- 7 ■ Leveraging the Model-View-Controller 166
- 8 ■ Testing JavaScript components 211
- 9 ■ REST API design and layered service architectures 251

4

Release, deployment, and monitoring

This chapter covers

- Understanding release flow and predeployment tasks
- Deploying to Heroku
- Using Travis for continuous integration
- Understanding continuous deployments

We've covered the build process, common build tasks you can perform (and how to do that using Grunt), and, at a high level, environments and configuration. We discussed the development environment extensively, but that's only half the story. The development environment is where you'll spend most of your time working, because you'll have a system in place, so you can prepare your application for a release, deploy it to a platform that humans can access, and then monitor the application state. Thanks to the Build First mentality, you'll be automating the workflows I've just mentioned, avoiding repetition, human error, and running tests, all while saving time, as I promised in chapter 1.

A continuous integration (CI) platform will help deploy more robust builds to production by ensuring your tests pass in a hosted environment. As you'll see later

in the chapter, CI helps test your code base remotely every time you push to your version control system (VCS). Build automation (and continuous development) is crucial for keeping your day-to-day development efforts productive and efficient. Comparably, having a workflow that's easy to execute ensures you can deploy your application as often as needed, without worrying about an embarrassing manual set of tasks that take half an hour to perform.

By the end of this chapter, you'll be ready to perform safe, continuous deployments, which are similar to continuous development in spirit. They're both intended to cut down the repetitive work and reduce human mistakes. The release flow has a few stages we're going to follow in this book:

- The first step is the build process, under the release distribution.
- Once the build is compiled, you'll run tests to make sure recent changes didn't break the build. Minor syntax issues should be constantly resolved during development by using lint programs.
- If the tests succeed, you might get into predeployment operations such as updating the version number and the release changelog.
- After that, you'll investigate deployment options, such as cloud hosting options and CI platforms.

Figure 4.1 describes this proposed release and deployment flow. As you look at the figure, keep a mental note of my proposal to deploy to staging first, to make sure everything works as expected in a hosted environment, before going live to production.

You have a long road ahead; let's commence by discussing the release and deployment flow. You'll visit predeployment operations in detail in section 4.2. Then in section 4.3, I'll tell you all about deployments, and you'll learn how to deploy an application to Heroku. Section 4.4 covers continuous integration and the tools you can use to get CI to do the heavy lifting on your behalf.

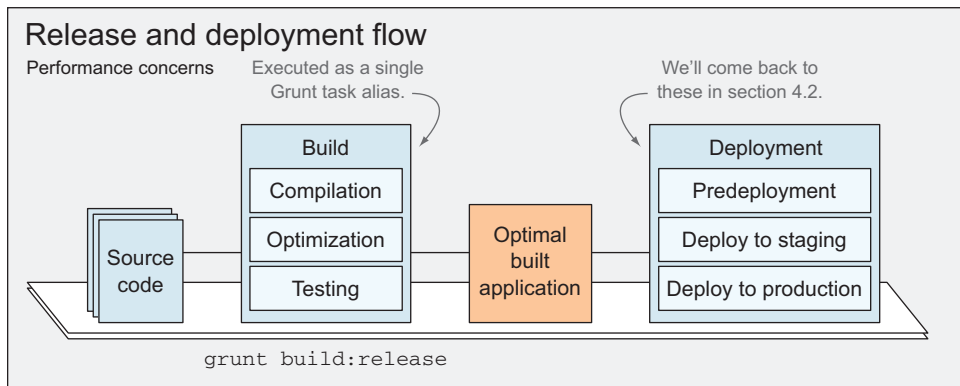


Figure 4.1 Proposed release and deployment flow

4.1 Releasing your application

When preparing your application for release, you'll want to place the web's best practices on your plate. In chapter 2, we discussed *minification*, shrinking your assets for better performance, and *concatenation*, joining files together to reduce the number of HTTP requests, which you'll definitely want to include in your release builds. These improve the web application's user experience by bundling your developer-readable source code into single files containing everything in the source code, but in a compressed form to hasten downloads. In that chapter we also covered *sprite maps and sprites*, large files containing many images. These would be used for debug distributions, too, for the sole reason that they enable you to keep debug and release more tightly bound together and less dissimilar. Otherwise you'd need to reference the individual icons in your debug CSS, and then somehow reference the spritemap and each icon's position in release, defeating the purpose of uniting both build flows and repeating yourself, breaking the DRY principle.

Minification, concatenation, spriting—what else is there to a release flow? In this section we'll go over image optimization and asset caching; then we'll move on to the deployment flow, semantic versioning, and keeping changelogs up-to-date effortlessly.

4.1.1 Image optimization

Concatenated and minified JavaScript and CSS files don't tell the whole story. Images represent, more often than not, the bulk of a web page's download footprint, meaning they are even more important to optimize than any other static assets. You already did a good chunk of optimization in chapter 2, when you examined how to generate a spritesheet using different images, which is comparable to how concatenation works for text files, merging many files into a single one. The other optimization, minification, reduces the contents of script and stylesheet files by shortening variable names and other micro-optimizations that minifiers perform. In the world of images, you have various ways to compress files, resulting in gains somewhere between 9% and 80%, typically above 50%. Luckily for us, certain Grunt packages, much like we're becoming accustomed to, do the heavy lifting for us in this regard.

One such package is `grunt-contrib-imagemin`, which does exactly what you want: image compression for different formats such as PNG, GIF, and JPG. Before plunging into it, I'll briefly cover the two aspects of image optimization it can help you with: lossless compression and interlacing.

LOSSLESS IMAGE COMPRESSION

Lossless image compression is, much like JavaScript minification, tasked with the removal of unimportant bits of data from your image's raw binary data. The important thing to notice is that lossless compression doesn't alter the image's appearance, but solely its binary representation. The only result of lossless compression is a smaller image that looks identical to the larger image. Lucky for us, smarter people have spent time working on tools that do advanced image compression work for us. You can specify the path to your image and have their algorithms work at it. Furthermore,

`grunt-contrib-imagemin` configures these low-level programs with the right parameters, so you don't have to. Note that lossless compression produces modest byte savings compared to lossy compression; it's great, however, when you can't afford to lose any image quality. When you can afford to lose image quality (and most of the time the losses are almost unnoticeable), you should use lossy image compression.

LOSSY IMAGE COMPRESSION

Lossy compression is an image compression technique where inexact approximation is applied (also known as partial data discarding) when re-encoding the image, resulting in far greater byte savings than those gained through lossless compression (up to 90% savings), where the removed information is usually only metadata such as geo-location, camera type, and so on. The `grunt-contrib-imagemin` package uses lossy compression by default, in addition to lossless compression, to remove unnecessary metadata. If you only want to use lossless compression, you should consider using the `imagemin` package directly.

INTERLACING IMAGES

The other image optimizing task you're going to study is *interlacing*.¹ Interlaced images have a larger size than regular images, but these added bytes are usually well worth it, because they improve perceived performance. Even though the image might take a little longer to complete downloading, it will start rendering faster than normal images do. Progressive images work exactly as they sound. They render a minimum view of the pixels in the image, which roughly looks like your complete image, and then they're progressively enhanced (as more data gets streamed to the browser), until the full-quality image is available.

Traditionally, images load top-down, in full quality, which translates into a faster download time but slower perceived rendering. The time to view the entire image equals the completion time. In progressive rendering mode, humans perceive a faster experience because they don't have to wait as long to see a (garbled) view of the entire image.

SETTING UP GRUNT-CONTRIB-IMAGEMIN

Setting up `grunt-contrib-imagemin` is, happily, as easy as the rest of the tasks we've gone over. Keep in mind that the important bits are in learning what the tasks do and how and when to apply them. The following listing configuration optimizes `*.jpg` images during release builds.

Listing 4.1 Optimizing images during release builds

```
imagemin: {
  release: {
    files: [{
      expand: true,
      src: 'build/img/**/*.jpg'
```

¹ Learn more about how interlacing improves perceived performance by visiting <http://bevacqua.io/bf/interlacing>. There's also an animated GIF that better explains how an interlaced image works.

```
  }},  
  options: {  
    progressive: true // progressive jpgs  
  }  
}  
}
```

Listing 4.1 doesn't need any extra configuration to compress the images; that's done by default. A fully working example can be found in the accompanying source for this chapter, labeled `ch04/01_image-optimization`, with a complete build workflow for both the `debug` and `release` distributions. Now that you've made the web a slightly better place for humans to drift around aimlessly, you can turn your attention to static asset caching.

4.1.2 Static asset caching

In case you're unfamiliar with the term, think of *caching* as photocopying history books from the library. Rather than going to the library every time you want to read them, you might prefer to print copies of a few pages, take those home, and read them whenever you please without having to hit the library again.

Caching in the web is more complicated than photocopying books borrowed from a library, but that should give you the gist of it.

EXPIRES HEADERS

A best practice you should definitely follow is using `Expires` headers for your static assets. This header, according to the HTTP protocol, tells the browser not to request the resource again if it was requested at least once (and therefore cached), and the cached version hasn't become stale. The expiration date in the `Expires` header determines when the cached version is no longer considered valid, and the asset has to be redownloaded. An example `Expires` header might be `Expires: Tue, 25 Dec 2012 16:00:00 GMT`.

This is both an awesome and a terrible practice. It's awesome for humans, because after their first visit to one of your pages, they don't need to redownload resources their browser stored in its cache, saving them requests and time. It's terrible for us, the developers, because it won't matter if you deploy changes to your assets, humans won't download them anymore.

To solve that inconvenience, and make `Expires` headers useful, you can rename your assets every time you deploy changes to them, appending a hash to their names, which forces browsers to download the file again, because for all intents and purposes, it's a different file from what they used to have in their cache.

HASHING A *hash* is a function that returns a fixed-length value that's an encoded representation of data. In your situation, the hash could be computed from the asset contents and its last modified date. One such hash might be `a38cbf9e`. Although seemingly arbitrary, there's no randomness involved. That would defeat the purpose of using an `Expires` header, because files would always have different names and be requested again every time.

Once you've computed a hash, you can use it as a query string parameter in your page, `/all.js?_a38cbf9e`, or you can append it to the filename, such as `/a38cbf9e.all.js`. Alternatively, you can add the hash to an `ETag` header. Choosing the right approach is a matter of identifying your needs. If you're dealing with static assets such as `JavaScript` resources, then you're probably better off hashing the filename (or its query string) and using an `Expires` header. If you're dealing with dynamic content, setting the hash in an `ETag` is preferred.

USING LAST-MODIFIED OR AN ETAG HEADER

An `ETag` header uniquely identifies one version of a resource. Similarly, `Last-Modified` identifies the last modification date of the resource. If you use either of these headers, then you should use the `max-age` modifier in the `cache-control` header, instead of the `Expires` header. This combination allows for softer caching, as the user agent can determine whether the cached copy should be used, or if the resource should be requested again. The following example shows how to combine the `ETag` and the `cache-control` headers:

```
ETag: a38cbf9e
Cache-Control: public, max-age=3600
```

The `Last-Modified` header behaves as an alternative to the `ETag` header, for convenience. Here we don't specify a uniquely identifying `ETag`, but achieve the same uniqueness by setting a modification date:

```
Last-Modified: Tue, 25 Dec 2012 16:00:00 GMT
Cache-Control: public, max-age=3600
```

Let's find out how you can use Grunt to create hashes for your file names that can then be used to set far-futures `Expires` headers safely.

CACHE BUSTING WITH GRUNT

Within your build process, you can do little to set HTTP headers, as those must go out with each response, rather than be statically determined. But what you can do is assign hashes to your assets using `grunt-rev`. This package will compute the hash for each of your assets and then rename them, appending the corresponding hash to their original names. For example, `public/js/all.js` would be changed to something such as `public/js/1be2cd73.all.js`, where `1be2cd73` would be the computed hash for the contents of `all.js`. One issue emerges from this task, and it's that now your views won't reference the correct assets, because they've been renamed with a hash in front of them. To remedy that, you can use the `grunt-usemin` package, which looks for static asset references in your HTML and CSS and refreshes them with the updated filenames. That's exactly what you need. The relevant Grunt configuration then looks like the following listing (labeled `ch04/02_asset-hashing` in the samples).

Listing 4.2 Updating filenames

```
rev: {
  release: {
    files: {
```

```

    src: ['build/**/*.css', 'build/**/*.js', 'build/**/*.png']
  },
  usemin: {
    html: ['build/**/*.html'],
    css: ['build/**/*.css']
  }
},

```

Keep in mind you don't have any use for either of these tasks in the `debug` flow, because these are optimizations that do nothing to benefit you during development, so it might be appropriate to name their targets `release` to make that distinction more explicit. The `usemin` task, however, is written in such a way that Grunt targets have a special meaning. The `css` and `html` targets are respectively used to configure which CSS and HTML files you want to update with the hashed filenames, but targets such as `release` would be ignored by `usemin`.

The next technique we'll cover involves inlining CSS in a style tag to avoid the render-blocking request for CSS, resulting in faster page loads.

4.1.3 Inlining critical above-the-fold CSS

Browsers block rendering whenever they encounter a CSS resource they need to download. Yet, we've taught each other for years to place CSS at the top of our pages (in the `<head>`), so users won't see a flash of unstyled content (abbreviated as FOUC). The inlining technique aims to improve page load time speed without damaging user experience by avoiding FOUC. This technique only works effectively if you're rendering your views on the server side as well as the client side, as we explore in chapter 7.

To implement this feature, you have to do a number of different things:

- First, you need to identify the “above-the-fold” CSS; these are the styles that are required to correctly render the visible elements on the page, on first load.
- Once we've identified the styles that are effectively used above the fold (those that the browser needs to render the page properly and avoid the FOUC), you need to inline them in a `<style>` tag on the `<head>` of your pages.
- Last, now that the required styles are inlined in a `<style>` tag, you can eliminate the render-blocking request for the CSS style sheet by deferring the request until after the `onload` event has triggered, using JavaScript.
- Naturally, you wouldn't want to leave users with JavaScript turned off stranded, and because we're good citizens of the web, you'll also use a fallback `<noscript>` tag to make the render-blocking request anyway.

As you've probably noticed, this is a complicated and error-prone process, much like the case study in chapter 1, where Knight's Capital lost half a billion dollars due to human error. It's probably not going to be that catastrophic for you if something goes wrong, but automating this process is almost mandatory: there's too much work involved to be done every time your styles change, or whenever your markup changes!

Let's learn how we can use Grunt to automate this process, using `grunt-critical`.

HAVING GRUNT DO THE HEAVY LIFTING

Using `grunt-critical` for this purpose is incredibly easy, although it does provide a wealth of configuration options. In the following code, you'll find the configuration for a simple use case. In this case, you're extracting critical CSS from a page and inlining those styles after the build, inside a `<style>` tag. `critical` goes the extra mile of deferring the rest of the styles so as not to block rendering, and it also adds the `<noscript>` fallback tag for those that have JavaScript disabled:

```
critical: {
  example: {
    options: {
      base: './',
      css: [
        'page.css'
      ]
    },
    src: 'views/page.html',
    dest: 'build/page.html'
  }
}
```

You probably are already familiar with all of the provided options, which are file paths. The `base` option indicates the root directory that should be used when finding absolute resource paths such as `/page.css`. Once you set up Grunt to perform inlining on your behalf, remember to serve the upgraded HTML files, rather than the prebuilt ones.

Before switching gears and soaking in the thermal spring waters of automated deployments, you need to reflect upon the importance of testing a release build ahead of each deployment to mitigate the possibility of the spring being in an active volcanic area.

4.1.4 Testing before a deployment

Before you get into the deployment stage, or even the predeployment stage as we'll explore soon, you need to test your release build. Testing a release build becomes important when there's a deployment in your future, because you want to make sure your application behaves as you expect, or at the least, behaves as the tests you've written expect it to behave.

In the next part of the book, we'll delve into the underworld of application testing and examine two types of testing (though many, many more exist) in detail. These are unit testing and integration testing:

- **Unit testing:** Here you test individual components of your application by isolating them, making sure the components work fine on their own.
- **Integration (or end-to-end) testing:** This takes a series of unit-tested components and tests the interactions between them, making sure they communicate appropriately.

It'll be a while before you embark on testing practices and examples. We'll discuss testing practices and see examples in chapter 8. Keep in mind that before deployments, you need to test your application, reducing the odds of shipping a faulty build to one of your hosted environments, particularly if said environment is production. Let's discuss a few more tasks you can perform after a release is tested but before it's deployed.

4.2 Predeployment operations

Once you've prepared a build for release and had it carefully tested, you're ready to deploy. But I have a couple of important predeployment tasks I want to mention before taking a swim in the deployment hot springs.

Figure 4.2 is an overview of the deployment flow, as well as the operations that come before a build can be considered deploy-ready. It also shows how you're going to progressively roll out your update to different environments, ensuring maximum predictability.

PREDEPLOYMENT OPERATIONS

- *Semantic versioning*: This helps keep track of meaningful application versions. Semantic versions are formatted similarly to **MAJOR.MINOR.PATCH-BUILD**. This standard helps avoid confusion when managing dependencies. Keeping your application versioned is important if you want any control over what code is currently deployed on hosted environments, such as production. It enables you to roll back to an older version when things go awry. Considering this is fairly easy to set up, and taking into account how costly it is to be unprepared for deployments not panning out, versioning becomes a no-brainer.
- *Change logging*: A *changelog* is a list of changes that were made throughout the history of your project, divided by which version they were introduced in (partly why keeping versions is important) and further segmented as bug fixes, breaking changes, and new features. By convention, changelogs in *git* repositories are often placed at the project root, and named something along the lines of

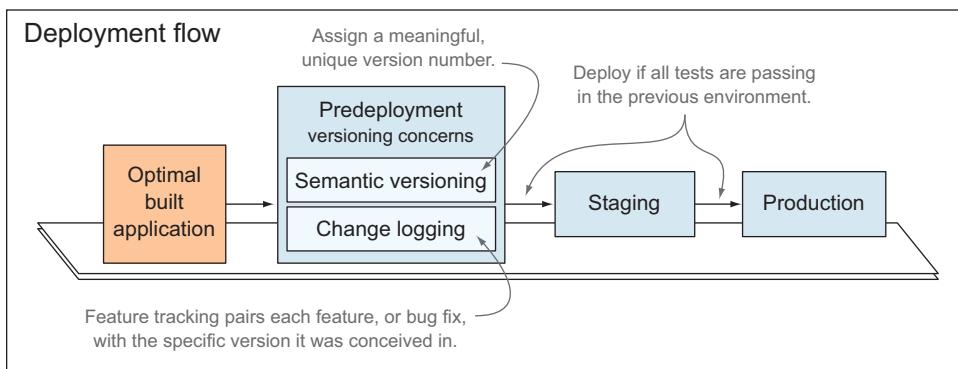


Figure 4.2 Versioning before a deployment and progressive deployment rollout. Testing by QA team in staging ensures robustness before deployment to production.

`CHANGELOG.txt`, or using whatever extension you prefer (such as `md` for Markdown,² a text-to-HTML conversion tool).

We'll delve into how you can better allocate your changelog upkeep time in a bit, but first let's explore the details of semantic versioning.

4.2.1 Semantic versioning

Because you're using Node, you might be familiar with the term semantic versioning. `npm` uses semantic versioning³ for all packages, because it's a powerful specification to manage dependency resolution among different Node modules. Because every Node application you produce already has a `package.json`, and considering those contain a semantic version in them, you'll use these to tag your releases before deployments.

When I talk about versioning, I mean updating the package version and then creating a tag (a moment in your version history you can refer to) in your VCS. You can set up any scheme you want when it comes to numbering your releases, but the important part is that you don't overwrite a release; you shouldn't make two releases using the same version number. To ensure this uniqueness, I've settled for increasing the build number after every build (regardless of distribution) automatically with Grunt, and I also increase the patch number when I perform a deploy. Major version changes are intentionally manual, as those are probably introducing breaking changes. The same applies for minor version changes, as new features are usually introduced in new minors.

With Grunt, you could perform these version increments (from now on referred to as bumps) using the `grunt-bump` package. It's easy to configure, it does the version tagging for you, and it even commits the changes to the `package.json` file for you. Here's an example:

```
bump: {  
  options: {  
    commit: true,  
    createTag: true,  
    push: true  
  }  
}
```

These are, in fact, the defaults provided by this task. They're sensible enough that you don't have to configure it at all. The task will bump the version found in `package.json`, commit exactly that file with a relevant message, and then create a tag in `git` to finally push those changes to the `origin` remote. If you turn off all three of those options, the task only updates your package version. Sample `ch04/03_version-bump` shows this behavior in action.

Once versioning is sorted out, you'll want to set up a changelog, enumerating what changed since the previous release. Let's mull that over.

² The Markdown format is a plain-text representation of HTML that's easy to read, write, and convert into HTML. Read the original article introducing Markdown in 2004 at <http://bevacqua.io/bf/markdown>.

³ You can read more about semantic versioning at <http://bevacqua.io/bf/semver>.

4.2.2 Using changelogs

You're probably used to reading changelogs from products that interest you when new releases come out (games, in particular, have a strong presence of changelogs in their culture), but have you ever maintained one yourself? It's not as hard as you might think.

Setting up a changelog—as an internal document that helps track changes made over time—could be a positive addition to your project even if you're not showing it to consumers.

If you have any sort of transparency policy, or you don't like keeping humans in the dark, then a changelog becomes almost mandatory to maintain. You shouldn't update the changelog every time you build for release, because you might want to produce a release build for debugging purposes. You shouldn't update them before testing, either. If testing fails, then the changelog would be out of sync with the last release-ready build. Then you're left with the need to update the changelog after you produce a build that passes all of the tests. Then and only then can you update the changelog to reflect the changes made since the last deployment.

Putting changelogs together is often hard because you forget what changed since the previous release, and you don't want to go through the `git` version history figuring out which changes deserve a spot in the changelog. Similarly, updating it by hand every time you make a change is tedious, and you might forget to do that if you're in the zone. A better alternative might be to set up `grunt-conventional-changelog` and have it build a changelog for you. All you'd have to do then is commit messages that, by convention, start with `fix` for bug fixes, `feat` when new features are introduced, or `BREAKING` when you break backwards compatibility. Furthermore, this package will allow you to edit the changelog by hand once it's done with its own parsing and updates.

As far as configuration goes, this task doesn't need any. Here are a few sample commit messages:

```
git commit -m "fix: buffer overflows, closes #17"
git commit -m "feat: reticulate splines for geodesic cape, closes #23"
git commit -m "feat: added product detail view"
git commit -m "BREAKING: removed POST /api/v1/users/:id/kill endpoint"
```

4.2.3 Bumping changelogs

The `bump-only` and `bump-commit` tasks allow you to bump the version without committing any changes, so that you can then update your changelog (as you'll see in a minute). Last, you should `bump-commit` to check in both `package.json` and `CHANGELOG.txt` at once in the same commit. Once you configure the `bump` task to also commit the changelog, you can now use the following alias to update your build version and changelog in one fell swoop. You can find an example using `grunt-conventional-changelog` in the samples, listed as `ch04/04_conventional-changelog`.

```
grunt.registerTask('notes', ['bump-only', 'changelog', 'bump-commit']);
```

Now you're done building for release, your tests are passing, and you've updated your changelog. You're ready to deploy to a hosted environment from which you can serve your application. In the past, it was fairly commonplace to deploy applications merely by means of uploading your built packages by hand to your production servers. You've come a long way from those good old days, and deployment tools, as well as application hosting platforms, have gotten better.

Let's next dive into Heroku, a Platform as a Service (PaaS) provider that enables you to deploy your application easily from the command line.

4.3 *Deploying to Heroku*

Setting up a deployment flow can be as hard as preparing sushi or as easy as ordering take-out; it all depends on how much control you want over the deployment. At one end of the spectrum you have services such as Amazon's Infrastructure as a Service (IaaS) platform, where you have full control over your hosted environment. You can pick your preferred operating system, choose how much processing power you'd like, configure it at will, install things on it, and then deal with the whole SysOps heavy lifting, such as securing the application against attacks, setting up proxies, picking a deployment strategy that guarantees uptime, and configuring most everything from the ground up.

On the other end of the spectrum are services where you don't have to do anything, such as those solutions often offered by domain name registrars such as GoDaddy. In these solutions you generally pick a theme, flesh out a few pages of static content, and you're done; everything else is done for you.

For the purposes of this book, I looked into the possibility of explaining how to host an application on Amazon, but I concluded that it'd be going too far off-scope. That being said, I'll be mentioning near the end of this section a way in which you can explore this alternative on your own.

I decided to go with Heroku (although there are similar alternatives, such as DigitalOcean), which isn't as complicated as setting up an instance on Amazon Web Services (AWS), but is fairly nontrivial, as opposed to using a website generator. Heroku simplifies your life by easily enabling you to configure and deploy your application to a hosted environment on their platform, straight from the command line. As I mentioned previously, Heroku is a Platform as a Service (PaaS) provider where you can host your application regardless of language or lack of server administration knowledge. In this section we'll go over the deployment of a simple application to Heroku, step by step.

At the time of this writing, Heroku offers a tier that allows you to host your applications with them for free. Let's get started there. You can find these instructions⁴ in the accompanying source code as well.

⁴ Find the Heroku deployment example online at <http://bevacqua.io/bf/heroku>.

- 1 Go to <https://id.heroku.com/signup/devcenter>, and enter your email.
- 2 The next manual step you need to follow is installing their toolbelt, a series of command-line programs that help you manage your applications hosted on Heroku. You can find it at <https://toolbelt.heroku.com>, and then follow the instructions to run `heroku login`, which you can find on that same website.
- 3 You'll then need a `Procfile`, which is a fancy file to describe the OS processes your application runs on.

Heroku's definition of a Procfile can be found below. Note that there are also a few more steps to this process that can be found a few paragraphs later.

PROCFILE A `Procfile` is a text file named `Procfile` placed in the root of your application that lists the process types in an application. Each process type is a declaration of a command that's executed when an instance (called `dyno` in Heroku's jargon) of that process type is started. You can use a `Procfile` to declare various process types, such as multiple types of workers, a singleton process like a clock, or a consumer of the Twitter streaming API.

Long story short, for most well-designed Node applications out there, the `Procfile` will look similar to the following code:

```
web: node app.js
```

As far as the application goes, you're going for the bare minimum, because this is a taste of what deploying to Heroku feels like. `app.js` could be as small as the following snippet of JavaScript (ch04/05_heroku-deployments):

```
var http = require('http');
var app = http.createServer(handler);

app.listen(process.env.PORT || 3000);

function handler (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('It\'s alive!');
}
```

Note that you use `process.env.PORT || 3000`, because Heroku will provide your application with a port it should listen on that will be exposed on the environment variable named `PORT`.

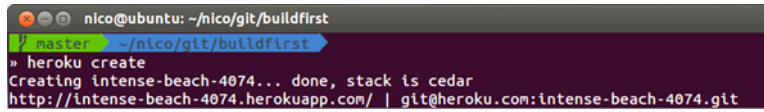
Then you use `3000` for local development. Now, here are a few more steps to take:

- 1 Once you're sitting on your project root, execute the following in terminal, to initialize a `git` repository:

```
git init
git add .
git commit -m "init"
```

- 2 Next create the app on Heroku with `heroku create`. This is a one-time thing.

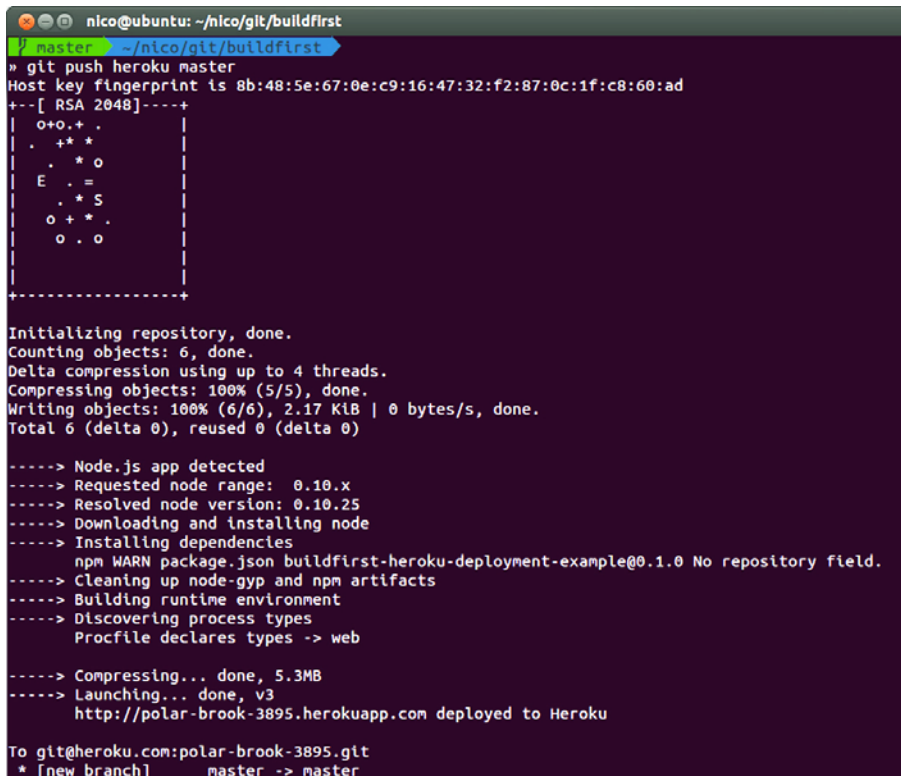
At this point, your terminal should look similar to figure 4.3.



```
nico@ubuntu: ~/nico/git/buildfirst
» master ~/nico/git/buildfirst
» heroku create
Creating intense-beach-4074... done, stack is cedar
http://intense-beach-4074.herokuapp.com/ | git@heroku.com:intense-beach-4074.git
```

Figure 4.3 Creating an app on Heroku using their CLI

On every deploy you want to make, you can push to the `heroku` remote using `git push heroku master`. This will trigger a deploy, which looks something like figure 4.4.



```
nico@ubuntu: ~/nico/git/buildfirst
» master ~/nico/git/buildfirst
» git push heroku master
Host key fingerprint is 8b:48:5e:67:0e:c9:16:47:32:f2:87:0c:1f:c8:60:ad
+--[ RSA 2048 ]-----+
|  O+O+.  |
|  +* *   |
|  . * O   |
|  E . =   |
|  . * S   |
|  O + * . |
|  O . O   |
+-----+
Initializing repository, done.
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 2.17 KiB | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)

-----> Node.js app detected
-----> Requested node range: 0.10.x
-----> Resolved node version: 0.10.25
-----> Downloading and installing node
-----> Installing dependencies
npm WARN package.json buildfirst-heroku-deployment-example@0.1.0 No repository field.
-----> Cleaning up node-gyp and npm artifacts
-----> Building runtime environment
-----> Discovering process types
Procfile declares types -> web

-----> Compressing... done, 5.3MB
-----> Launching... done, v3
http://polar-brook-3895.herokuapp.com deployed to Heroku

To git@heroku.com:polar-brook-3895.git
* [new branch] master -> master
```

Figure 4.4 Deploying to Heroku—as simple as `git push`

If you want to pull up the application in the browser, use the following command:

```
heroku open
```

There's one caveat about Heroku and PaaS providers. When it comes to deploying build results, there's no simple solution. You shouldn't include build artifacts in your repository, as that may cause undesirable results such as forgetting to rebuild after

changing something. You shouldn't get too comfortable building on their platforms, either, because building is something that should be done locally or on an integration platform, but not on the application server itself, because that would put a dent in your application's performance.

4.3.1 Deploying builds

The problem is you shouldn't put build results in version control, because those are the output of your source. Instead you should build before deployments, and deploy the build results along with the rest of your code. Most PaaS providers don't offer many alternatives. Platforms such as Heroku take deployments from Git when you push to their remote, but you don't want to include the build artifacts in revision control, so that becomes an issue. The solution: treat Heroku as you would any continuous integration platform (more on that in section 4.4), and allow Heroku to build your application in its servers.

Heroku doesn't usually install `devDependencies` for Node projects, because it uses `npm install --production`, and you need to use a custom buildpack to get around that. *Buildpacks* are interfaces between the language you use and the Heroku platform, and they're collections of shell scripts. Creating an application with the custom Grunt-enabled buildpack is easy using the following command, where `thing` is the name of your app on Heroku:

```
heroku create thing --buildpack https://github.com/mbuchetics/heroku-  
buildpack-nodejs-grunt.git
```

Once you've created an application using the custom buildpack, you could push the way you usually do, and that would trigger a build on Heroku servers. The last thing you need to set up is a `heroku` task:

```
grunt.registerTask('heroku', ['jshint']);
```

Heroku will terminate deployments if the build fails, keeping the previously deployed application unaffected by failed builds. There's a detailed explanation in the accompanying samples, listed as `ch04/06_heroku-grunt`, which will walk you through setting this up.

Let's take a look at how you can fit multiple environments in a single Heroku application.

4.3.2 Managing environments

If you want to set yourself up so you can host multiple environments⁵ on Heroku, such as `staging` and `production`, use different `git` remote endpoints to achieve this. Create a remote other than `heroku` with the CLI:

```
heroku create --remote staging
```

⁵ Heroku has advice on managing multiple environments. Go to <http://bevacqua.io/bf/heroku-environments>.

Instead of `git push heroku master`, you should now do `git push staging master`. Similarly, instead of doing `heroku config:set FOO=bar`, you now need to explicitly tell `heroku` to use a particular remote, such as `heroku config:set FOO=bar --remote staging`. Remember environment configuration is environment-specific, and should be treated as such, so environments shouldn't share API keys to third-party services, database credentials, or any authentication data in general.

Now that you can configure and deploy to specific environments directly from your command line, it's time to learn about a practice known as continuous integration, which will help tighten the leash on overall code quality. If you want to look into deployments to Amazon Web Services, there's a small guide⁶ you can follow in the accompanying source code (labeled `ch04/07_aws-deployments` in the samples).

4.4 *Continuous integration*

Martin Fowler is one of the most renowned proponents of continuous integration. In his own words,⁷ Fowler describes CI as follows.

CONTINUOUS INTEGRATION is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily, leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Furthermore, he entices us to run the test suite in an environment that's as close to our production environment as possible. The implication is that your best bet, when it comes to testing your application, is doing it in the cloud, the way you do your hosting. CI platforms such as Travis-CI provide features like build error notifications and access to the full build logs, detailing everything that happened during the build (and its testing).

I mentioned Travis-CI; let's see how we can set ourselves up in such a way that we can remotely add builds to a queue on its platform on every commit made to our repository. Then Travis-CI build servers will process this queue one item at a time, running our builds and letting us know about the results.

4.4.1 *Hosted CI using Travis*

Continuous integration means to run tests on a remote server (which is as similar as possible to the production environment) in hopes of catching bugs that would otherwise make their way to the general population. Travis-CI is one CI platform (Circle-CI is another) where you can get feedback remotely on the result of a build once you've properly configured it. If the build is successful, you won't even notice. If the build

⁶ Walk through the deployment process to AWS with this code sample at <http://bevacqua.io/bf/aws>.

⁷ Read Fowler's full article on continuous integration at <http://bevacqua.io/bf/integration>.

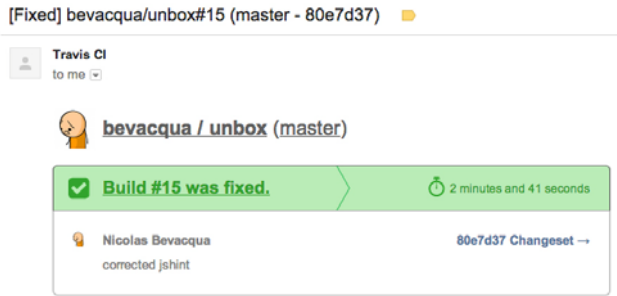


Figure 4.5 A typical Travis notification for a build fix

fails, you'll get an email notification telling you someone broke your build (oops!). Later, when a subsequent push fixes the build, you'll get another notification letting you know about the fix. Additionally, you can also access full build logs on the Travis website, which always comes in handy when figuring out why a build failed. Figure 4.5 shows one such email notification.

Setting up CI is almost too easy in this day and age. The first thing you'll need to do is create a `.travis.yml` file at the project root. In the file, you'll need to declare the language you're using, which in your case is identified as `node_js`, the runtime version you're testing your builds against, and a series of scripts to execute before, during, and after the integration test. For the purposes of illustration, such a file might look like the following code:

```
language: node_js

node_js:
  - "0.10"

before_install:
  - npm install -g grunt-cli  script:    - grunt ci --verbose --stack
```

CONFIGURING TRAVIS AND GRUNT

Before executing your tests, you need to install the command-line interface for Grunt, `grunt-cli`, through `npm`. You'll need it in the integration test server the way you need it in your development environments so you can run Grunt tasks. You can use the `before_install` section to install the CLI.

All that's left then is to set up a `ci` task for Grunt. The `ci` task could run `jshint` to mitigate syntax errors, just like you're already doing locally every time something changes, thanks to your newfangled continuous development workflow. You should configure the `ci` task to run unit and integration tests as well, on top of linting your code with `jshint`.

The real value in CI comes from having the remote server build your entire application and apply your tests (lint included) against the code base, ensuring you don't depend on files not checked into version control or dependencies you might have installed locally but not made available in your code base at large.

You'll probably want to try out this example yourself, and I recommend you do so, because it's a good exercise for deployment-craving minds. You can follow the

detailed instruction set I laid out in the accompanying sample repository,⁸ named 08_ci-by-example, under ch04. Once you're done with that, you might as well learn about continuous deployments, a practice that may or may not fit into your workflow, but one that you should be fully aware of, regardless.

4.4.2 Continuous deployments

The Travis platform supports continuous deployments to Heroku.⁹ *Continuous deployments* are a fancy way of saying that every single time you push to version control, you also trigger a build job in the CI server (which you're already doing as of last section, when you turned on Travis CI integration). When those builds succeed, the CI server deploys on your behalf to the release environments of your choosing.

In my experience, continuous deployments are a two-edged sword. When they work, you are cutting into a world of joy and less tedious deployments where passing the build and test integration cycle is validation enough to push to production. But you have to be confident that you've got enough tests in place to catch errors sensibly. A safe bet might be to enable continuous deployment to your staging environment rather than directly to production. Then, you'd make sure there are no issues in staging, and perform a deploy to production. This workflow looks like figure 4.6.

There's work involved in enabling continuous deployments to Heroku. You need an API key from Heroku, and you need to encrypt it and then configure `.travis.yml` with the encrypted data. I'll leave that up to you, now that I've voiced my concerns about deploying to production directly. If you choose to do that, visit <http://bevacqua.io/bf/travis-heroku> for instructions.

We've spent the majority of this chapter addressing deployments, which is a good thing. Now you can finally turn your attention to the options you have when it comes

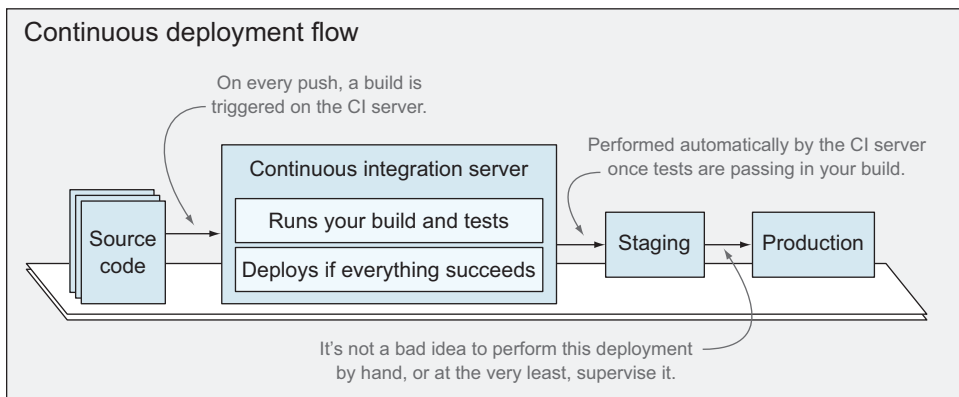


Figure 4.6 Proposed continuous deployment flow

⁸ Find the fully documented code sample online at <http://bevacqua.io/bf/travis>.

⁹ Read the article on Travis continuous deployments to Heroku at <http://docs.travis-ci.com/user/deployment/heroku/>.

to monitoring the state of your application as a whole, and individual requests in particular, when live in production. You'll also examine approaches to logging, debugging, and catastrophe tracing.

4.5 Monitoring and diagnostics

Production application monitoring is as important as having loyal customers. If you don't appreciate application uptime, your customers won't appreciate you. This is to say that you can't afford not to monitor your production servers. By monitoring I mean keeping access logs (who's visited what, when, and where from), as well as error logs (what went wrong), and perhaps even more importantly, setting up alerts so that you are immediately notified when things go *expectedly* wrong. "Expectedly" wasn't a typo; you should expect things to go wrong, and be as prepared as you can for those situations. Your enterprise probably doesn't warrant a simian army roaming around and randomly terminating off instances and services like Netflix advocates¹⁰ to ensure their servers can reliably and consistently endure faults, such as hardware failure, without it affecting the end users consuming their services. But their advice, quoted as follows, still applies to most every software development effort.

QUOTE FROM NETFLIX BLOG If we aren't constantly testing our ability to succeed despite failure, then it isn't likely to work when it matters most—in the event of an unexpected outage.

How do you plan for failure, though? Well, that's the sad part; nothing you do will prevent failure. Everyone has downtime, even giants such as Microsoft, Google, Facebook, and Twitter. You can plan all you want, but your application is going to fail regardless of what you do. What you can do is develop a modular architecture that's capable of dealing with services going boom and instances going bust. If you can achieve that modularity, it shouldn't be as damaging when a single module stops working, because the rest would still be perfectly functional. We'll develop notions of modularity, and the single responsibility principle (SRP) in chapter 5, dedicated to modular design and a crash-course introduction to the Node.js platform.

The first rule about Fight Club is you do not talk about Fight Club. Sorry, wrong movie. The first rule about application monitoring is you log things and set up notifications when bad things happen. Let's go over a possible implementation for that.

4.5.1 Logging and notifications

I'm sure you're more than used to `console.log` on the front end to inspect variables, and maybe even as a debugging mechanism, using it to figure out which code paths are being followed, and helping you nail down bugs. On the server side you have the standard output and standard error streams, both logging to your terminal window. These transports (`stdout` and `stderr`; more on transports in a minute!) are useful for

¹⁰ Learn about Chaos Monkey, a chaos mongering service at Netflix, at <http://bevacqua.io/bf/netflix>.

development, but they are near useless to you if you can't capture what's being transmitted to them in a hosted environment, where you can't monitor the process in your own terminal.

Heroku has a mechanism where it captures the standard output of your processes, so you can access it down the road. It also has add-ons to further extend that behavior. Heroku add-ons provide much-needed companion services such as databases, emailing, caching, monitoring, and other resources. Most logging add-ons would allow you to set up filtering and notifications; however, I'd advise against leveraging Heroku's logging capabilities, as that would be too platform-specific, and it can severely limit your ability to migrate to another PaaS provider. Dealing with logging on your own isn't that hard, and you'll soon see the upside of doing so.

WINSTON FOR LOGGING

I'm not a huge fan of taking advantage of the Heroku logging facilities, because it binds your code base to their infrastructure by assuming writing to standard output will suffice in your log tracking efforts. A more durable and versatile approach would be to use a multitransport logger rather than writing to `stdout`. Transports dictate what happens with the information you're trying to log. A transport might log to a file, write a database record, send an email, or send push notifications to your phone. In multitransport loggers, you can use many of these at the same time, but you'd still use the same API to perform the logging. Adding or removing transports doesn't affect the way you write log statements.

Node has a few popular logging libraries, and I've picked `winston` because it has every feature you're looking for in a logger: logging levels, contexts, multiple transports, an easy API, and community support. Plus, it's easily extensible, and people have written transports for nearly everything you'll ever need.

By default, `winston` uses the `Console` transport, which is the same as using `stdout` directly. But you can set it up to use other transports, such as logging to a database or to a log management service. The latter are notably flexible in that they provide a platform where you can choose to get notified on important events without changing anything in your application.

Using a logging solution such as `winston` is platform agnostic. Your code won't depend on the hosting platform to capture standard output to work. To get started using `winston`, you have to install the package by the same name:

```
npm install --save winston
```

USING `--save` VS USING `--save-dev`

In this case, you'll use the `--save` flag rather than `--save-dev`, because `winston` isn't a build-only package like the Grunt packages you've toyed with so far. When providing the `--save` flag to `npm`, the package will be added to your `package.json` file under `dependencies`.

Once you've installed `winston`, you can use it right away by putting `logger` where you used to put `console`:

```
var logger = require('winston');

logger.info('east coast clear as day');
logger.error('west coast not looking so hot.');
```

You might have gotten used to the idea of `console` being a global variable. In my experience, it's not wrong to use globals in this kind of scenario, and it's one of the two cases where I allow myself to use globals (the other one being `nconf`, as I mentioned in chapter 3). I like setting all the globals in a single file (even if there are only two), so that I can quickly scan it and figure out what's going on when I call something that's not otherwise defined in a module, or a part of Node. An illustrative `globals.js` might be as follows:

```
var nconf = require('nconf');

global.conf = nconf.get.bind(nconf);
global.logger = require('./logger.js');
```

I also propose keeping a single file where you can define the transports for your logger. Let's kick things off by using a `File` transport, as well as the default `Console` one. This would be the `logger.js` file referenced in the previous snippet:

```
var logger = require('winston');
var api = module.exports = {};
var levels = ['debug', 'info', 'warn', 'error'];

levels.forEach(function(level) {
  api[level] = logger[level].bind(logger);
});

logger.add(logger.transports.File, { filename: 'persistent.log' });
```

Now, whenever you do `logger.debug`, you'll be logging a debug message to both the terminal and to a file. Although convenient, other transports offer more flexibility and reliability, and such is the case of a few transports we'll be covering in the accompanying samples: `winston-mail` will enable you to send out emails whenever something happens (at a level that warrants an email), `winston-pushover` sends notifications directly in your phone, and `winston-mongodb` is one of many traditional logging transports where you write a record in your database.

Once you've made sure to check out the sample listings, you'll have a better idea of how configuration, logging, and globals are tied together according to what I suggested. In case you're religiously against globals, don't panic. I've also included a sample where they aren't used. I like globals (in the two cases I mentioned previously) only because I find it convenient not having to `require` the same things in every module.

Now that you've spent time dealing with logging, we might as well talk about debugging Node applications.

4.5.2 Debugging Node applications

You'll want all the help you can get when it comes to tracing down a bug, and in my experience the best approach to debugging is increased logging, which is one of the reasons we've talked about it. That being said, you have more than a few ways to debug Node apps. You might use `node-inspector`¹¹ inside of Chrome's DevTools, you could use the features provided by an integrated IDE such as WebStorm, and then there's good old `console.log`. You could also use the native debugger¹² in V8 (the JavaScript engine Node runs on) directly.

Depending on which kind of bug you're tracing, you'll pick the right tool for the job. For example, if you're tracing a memory leak, you might use a package such as `memwatch`, which emits events when it's likely that a memory leak occurred. A more common use case, such as pinning down a rounding bug, or finding out what's wrong with your API calls, can be satisfied by adding log statements (temporarily with `console.log`, or in a more permanent fashion with `logger.debug`), or using the `node-inspector` package.

USING NODE INSPECTOR

The `node-inspector` package hooks onto the native debugger in V8, but it lets you debug using the full-featured debugging tools found in Chrome as an alternative to the terminal-based debugger provided by Node. To use it, the first thing you'll need to do is install it globally:

```
npm install -g node-inspector
```

To enable debugging on your Node process, you can pass the `--debug` flag to `node` when you launch the process, like so:

```
node --debug app.js
```

As an alternative, you can enable it on a running process. To do this, you'll need to find the process ID (PID). The following command, `pgrep`, takes care of that:

```
pgrep node
```

The output will be the PID for your running Node process. For example, it might be as follows:

```
89297
```

Sending a `USR1` signal to the process will enable debugging. This is done using the `kill -s` command (note I'm using the process ID from the results of the previous command):

```
kill -s USR1 89297
```

¹¹ Find the open source repository for `node-inspector` at GitHub at <http://bevacqua.io/bf/node-inspector>.

¹² Read the Node.js API documentation on debugging at <http://bevacqua.io/bf/node-debugger>.

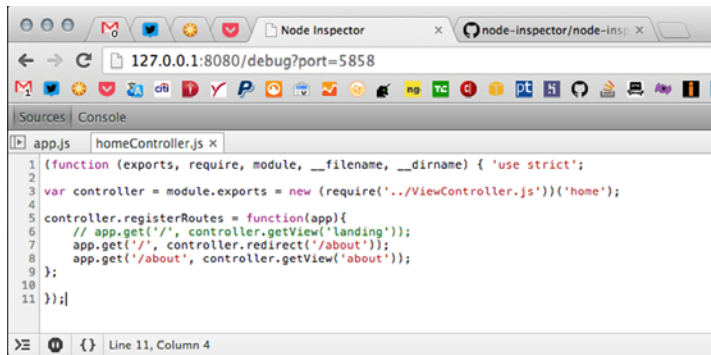


Figure 4.7 Debugging Node.js code in Chrome using Node Inspector

If everything worked correctly, Node will notify you where the debugger is listening through its standard output:

```
Hit SIGUSR1 - starting debugger agent.
debugger listening on port 5858
```

Now you need to execute `node-inspector` and then open Chrome, pointing it at the link provided by the inspector:

```
node-inspector
```

If all goes well, you should see something similar to figure 4.7 and have a full-blown debugger in your Chrome browser ready to use, which will behave (for the most part) exactly like the debugger for client-side JavaScript applications. This debugger will allow you to watch expressions, set breakpoints, step through the code, and inspect the call stack, among other useful features.

On a higher level than debugging, there's performance analysis, which will help detect potential problems in your code, such as memory leaks causing a spike in memory consumption that could cripple your servers.

4.5.3 Adding performance analytics

You have a few options when it comes to performance profiling, depending on how specific (we must track down a memory leak!) or generic (how could we detect a spike in memory consumption?) your needs are. Let's look into a third-party service, which can relieve you of the burden of doing the profiling on your own.

Nodetime is a service you can literally set up in seconds, which tracks analytics such as server load, free memory, CPU usage, and the like. You can sign up at <http://bevacqua.io/bf/nodetime-register> with your email, and once you do you'll be provided with an API key you can use to set up `nodetime`, which takes a few lines of JavaScript to configure:

```
require('nodetime').profile({
  accountKey: 'your_account_key',
  appName: 'your_application_name'
});
```

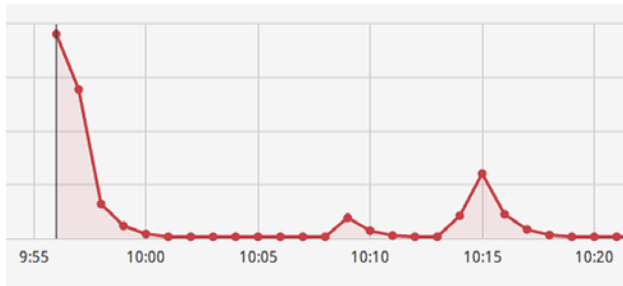


Figure 4.8 Server load over time, tracked by Nodetime

That's it, and you'll now have access to metrics, as well as the ability to take snapshots of CPU load, like the one presented in figure 4.8.

To conclude, we'll analyze a useful process scaling technique available to Node applications, known as `cluster`.

4.5.4 Uptime and process management

When it comes to release environments, production in particular, you can't afford to have your process roll over and die with any particular exception. This can be mitigated using a native Node API called `cluster` that allows you to execute your application in multiple processes, dividing the load among them, and create new processes as needed. `cluster` takes advantage of multicore processors and the fact that Node is single-threaded, allowing you to easily spawn an array of processes that run the same web application. This has the benefit of making your app more fault tolerant; you can spawn a new process! For example, in a few lines of code, you could configure `cluster` to spawn a worker every time another one dies, effectively replacing it:

```
var cluster = require('cluster');

// triggered whenever a worker dies
cluster.on('exit', function () {
  console.log('workers are expendable, bring me another vassal!');
  cluster.fork(); // spawn a new worker
});
```

This doesn't mean you should be careless about what happens inside your processes, as starting new ones can be expensive. Forking has a cost, tied to the amount of load your servers are under (requests / time), and also tied to the *startup time* for your process (wait period between spawning it and when it can handle HTTP requests). What `cluster` gives us is a way to transparently keep serving responses even if your workers die: others will come in his name.

In chapter 3 we introduced `nodemon` as a way to reload your application whenever a file changed during active development. This time you'll review `pm2`, which is similar to `nodemon` in spirit, but geared toward release environments.

ARRANGING A CLUSTER

Configuring `cluster` can be tricky, and it's also an experimental API at this time, so it might change in the future. But the upsides brought forth by the `cluster` module are

undeniable and definitely appealing. The `pm2` module allows you to use fully configured `cluster` functionality in your application without writing a single line of code, making it a no-brainer to use. `pm2` is a command-line utility, and you need to install it using the `-g` flag:

```
npm install -g pm2
```

Once installed, you can now run your application through it, and `pm2` will take care of setting up `cluster` for you. Think of the following command as a drop-in replacement for `node app`:

```
pm2 start app.js -i 2
```

The main difference is that your application will use `cluster` with two workers (due to the `-i 2` option). The workers will handle requests to your app, and if one of them crashes, another one will spawn so that the show can go on. Another useful perk of `pm2` is the ability to do *hot code reloads*, which will allow you to replace running apps with their newly deployed counterpart without any downtime. You'll find related examples in the accompanying source code, listed as `ch04/11_cluster-by-pm2`, as well as one on how to use `cluster` directly, listed as `ch04/10_a-node-cluster`.

While clustering across a single computer is immediately beneficial and cheap, you should also consider clustering across multiple servers, mitigating the possibility of your site going down when your server crashes.

4.6 Summary

Phew, that was intense! We worked hard in this chapter:

- You became more intimate friends with release flow optimizations such as image compression and static asset caching.
- You learned about the importance of testing a release before calling it a day, bumping your package version, and putting together a changelog.
- Then you went through the motions of deploying to Heroku, and I mentioned `grunt-ec2`, which is one of many alternative deployment methods.
- Attaining knowledge on continuous integration was a good thing, as you've learned the importance of validating your build process and the quality of the code base you released.
- Continuous deploys are something you can perform, but you understand the implications of doing that, so you'll be careful about it.
- You also took a quick look at logging, debugging, managing, and monitoring release environments, which will prove fundamental when troubleshooting production applications.

All this talk about monitoring and debugging calls for a deeper analysis of architecture design, code quality, maintainability, and testability, which are conveniently at the core of part 2 in the book. Chapter 5 is all about modularity and dependency management, different approaches to JavaScript modules, and part of what's coming in ES6 (a long

awaited ECMAScript standard update). In chapter 6, you'll uncover different ways you can properly organize the asynchronous code that's the backbone of Node applications, while playing it safe when it comes to exception handling. Chapter 7 will help you model, write, and refactor your code effectively. We'll also analyze small code examples together. Chapter 8 is dedicated to testing principles, automation, techniques, and examples. Chapter 9 teaches you how to design REST API interfaces and also explains how they can be consumed on the client side.

You'll leave part 2 with a deep understanding of how to design a coherent application architecture using JavaScript code. Pairing that with everything you've learned in part 1 about build processes and workflows, you'll be ready to design a JavaScript application using a Build First approach, the ultimate goal of this book.

JavaScript Application Design

Nicolas Bevacqua

The fate of most applications is often sealed before a single line of code has been written. How is that possible?

Simply, bad design assures bad results. Good design and effective processes are the foundation on which maintainable applications are built, scaled, and improved. For JavaScript developers, this means discovering the tooling, modern libraries, and architectural patterns that enable those improvements.

JavaScript Application Design: A Build First Approach introduces techniques to improve software quality and development workflow. You'll begin by learning how to establish processes designed to optimize the quality of your work. You'll execute tasks whenever your code changes, run tests on every commit, and deploy in an automated fashion. Then you'll focus on designing modular components and composing them together to build robust applications.

What's Inside

- Automated development, testing, and deployment processes
- JavaScript fundamentals and modularity best practices
- Modular, maintainable, and well-tested applications
- Master asynchronous flows, embrace MVC, and design a REST API

This book assumes readers understand the basics of JavaScript.

Nicolas Bevacqua is a freelance developer with a focus on modular JavaScript, build processes, and sharp design. He maintains a blog at ponyfoo.com.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/JavaScriptApplicationDesign

“Enjoy the ride through the process of improving your development workflow.”

—From the Foreword by
Addy Osmani, Google

“For JavaScript developers, a must-read!”

—Stephen Wakely
Thomson Reuters

“An excellent guide through the maze of the modern JavaScript ecosystem.”

—Jonas Bandi, IvoryCode GmbH

“The first-ever design book for developers.”

—Sandeep Kumar Patel, SAP Labs

“A one-stop shop introducing JavaScript developers to modern practices and tools.”

—Matthew Merkes, MyNeighbor



ISBN 13: 978-1-617291-95-1
ISBN 10: 1-617291-95-1



9 781617 129195 1