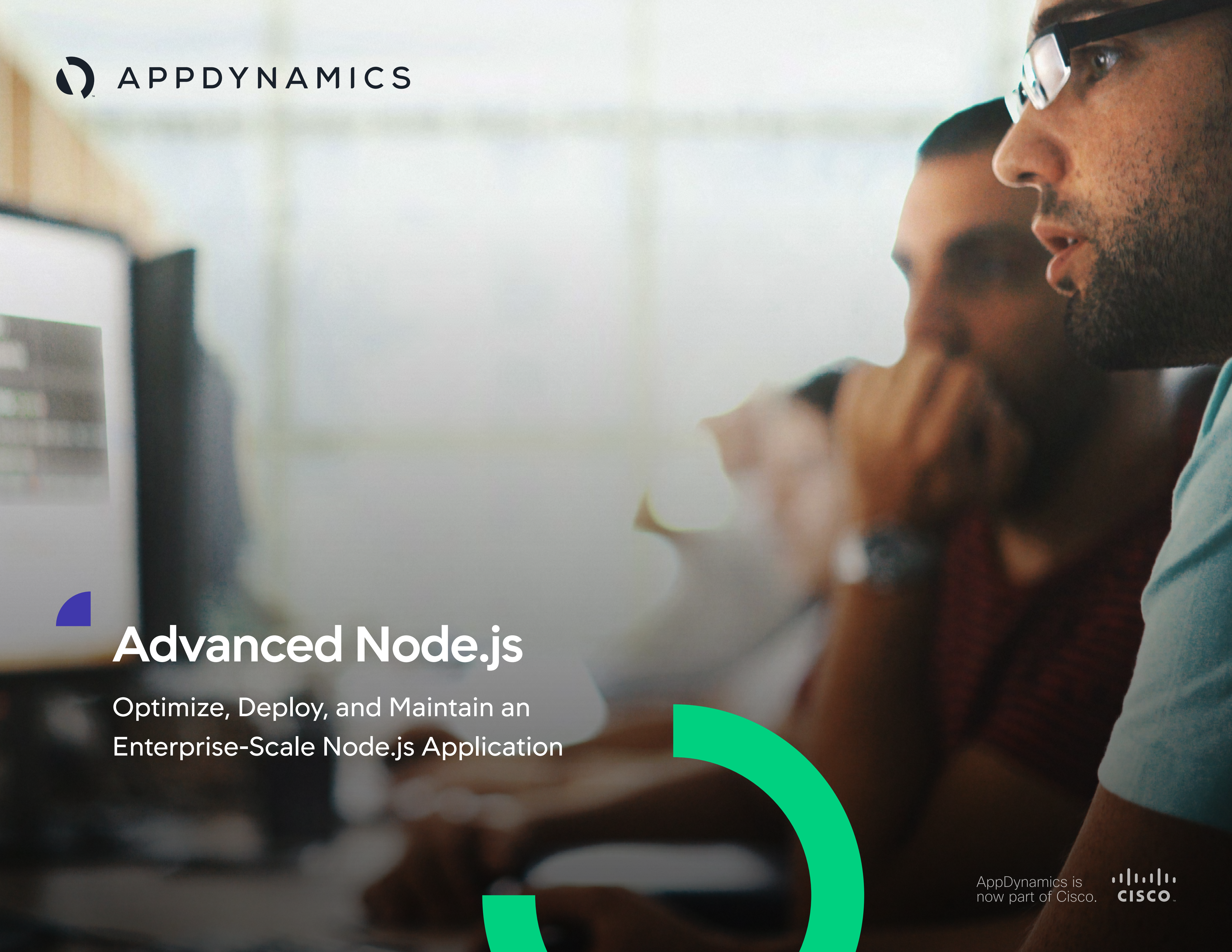
APPDYNAMICS



Advanced Node.js

Optimize, Deploy, and Maintain an Enterprise-Scale Node.js Application

AppDynamics is now part of Cisco.

Introduction:
Charting Your Path to Real-World Success Using Node.js

Less than a decade after its initial release, Node.js has become a pivotal technology for building enterprise-grade web applications. At the time, Node.js addressed a growing need for a way to build fast and scalable server-side applications. Today, the explosion of dynamic, responsive, data-driven online content and applications has turned this need into an absolute necessity.

If you're reading this introduction, you probably don't need a detailed walkthrough of what Node.js can do, how to get started using it, or why it has become a core server-side technology within some of the world's biggest corporations. *In fact, we're going to avoid these topics entirely.* There are already many excellent resources available to introduce novice developers to Node.js and the world of server-side JavaScript coding, so there's no point in covering the same ground.

Where we do want to focus our attention — and yours — is on a very different and very important topic: How to launch and run an enterprise-scale product, service, or brand built on Node.js.

This is a topic that has not, in our opinion, gotten the attention and expert insights it deserves. In most cases, this post-launch journey is far longer and has a bigger impact than the development process itself. It is also where a Node.js application will succeed or fail at translating promises and potential into real-world value, relevance, and business impact. At different points in this journey, the leader of a development team will call upon an arsenal of skills and strategies — learning how to maximize project efficiency, scale and manage growth, anticipate and address security risks, balance cost and complexity, and perform many other tasks, as well.

What This eBook Will Provide

In the pages that follow, we'll give you a practical foundation for success during the critical first three months or so of a successful enterprise-scale Node.js journey. This time span covers the period from pre-production planning to continuous deployment and testing — often in an environment that demands massively scaling your codebase, team, and audience.

In addition to covering the key tools and techniques you'll need at various points in this journey, we'll offer guidance in the form of Node.js best practices and case studies — building your success upon other teams' experiences.

This eBook isn't meant to cover every one of these topics completely or to create an exhaustive Node.js technical reference. Our goal is to give readers enough context and detail to understand the issues, gain basic competence in dealing with them in real-world situations, and to set them up for long-term success in building and deploying Node.js applications.

Before you get started, we recommend reviewing the table of contents to get a "lay of the land" overview. Good luck, and enjoy the process of mastering the entire Node.js journey.

Table of Contents

Chapter 1
The Journey Begins: Preparing for Production Launch

Preparing for a release is always a critical point in any application development journey, and that's certainly the case for Node.js projects. It's your team's final opportunity to find and fix issues before they impact your deployment process, your end users, or the business itself. While errors can and will impact your Node.js production environment just as they will any other type of software, our goal is to outline a process that minimizes the risk of avoidable issues while also striking the right balance between efficiency, cost, and use of team resources.

In this chapter, we'll walk you through a pre-release process with the following areas of emphasis:

- Optimizing Your Code
- Best Practices for Error Handling
- Confirming Your Code Meets Security Requirements
- Configuring for a Production Environment
- Deployment Considerations

Along the way, we'll share our recommendations on tools, summarize key best practices, alert you to common issues that may impact your codebase or deployment process, and link out to additional information when appropriate.

Optimizing Your Code

While this eBook is not intended to be a coding guide for Node.js, there are some highly desirable coding practices that can help to optimize your Node.js deployment.

Node.js is NOT a Web Server

Node.js was never meant to function as a web server, so if you set up your deployment to listen in on an HTTP port, this is a recipe for trouble. Node.js will never function as efficiently as NGINX or Apache, for example, so don't deploy it this way. Instead, configure your web servers to proxy connections to your Node.js instances to avoid the risk of turning your Node.js deployment into a performance bottleneck.

Don't Use Node.js to Serve Static Assets

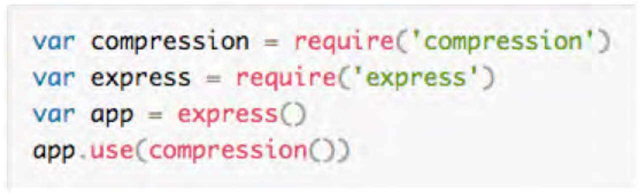
Although it's quite possible to have Node.js serve static content, it is very inefficient and uses a lot of memory. Have your web servers handle static content and make use of a CDN so that static content is served by geolocation.

Don't Use Synchronous Methods to Serve Requests

Node.js doesn't work well in synchronous mode, so always use Asynchronous functions. For example, fs.readFile(err, fileContent) instead of fs.readFileSync(function(err, fileContent) {});

Use gzip Compression

gzip compression can greatly increase the speed of your web application. This is an example of how to use the compression expression to your advantage:



```
var compression = require('compression')
var express = require('express')
var app = express()
app.use(compression())
```

Linting

The best way to carry out a final code optimization is to run an automated code quality tool through your codebase. This process is called "linting" and usually covers only very basic quality issues, but that's the point: It catches avoidable — and usually very easy-to-fix errors — before they put a production application at risk.

Besides preventing bugs, linting also ensures adherence to coding standards, promotes better collaboration, identifies any undeclared variables, and enables less dependence on other members of the development team. ESLint and JShint are two popular linting tools for Node.js code. We cover ESLint in more detail below.

ESLint

ESLint is an open-source JavaScript linting tool. It works by scanning the codebase and applying rules that define established stylistic guidelines. ESLint comes with some built-in rules, but the program also allows you to load your own rules at any time and create custom rules that align with a specific framework. Developers can configure ESLint to assign two severity levels to codebase issues: Warning (less severe) and Error (more severe).

Best Practices for Linting

The linting process can apply many coding standards. The ESLint website includes a list of key rules, grouped by category. These include rules dealing with possible errors (JavaScript syntax or logic errors) and with best practices (better ways of doing things within a codebase). By default, ESLint does not enable any rules, leaving it up to the developer to choose an appropriate set of rules for a given project.

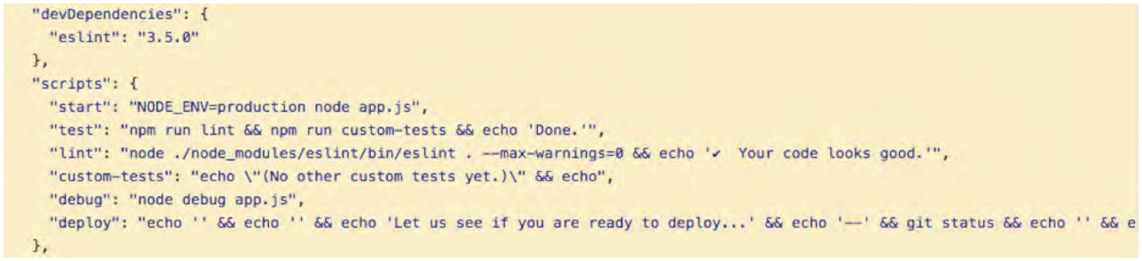
Flexible ESLint Configuration

There are two ways to configure ESLint:

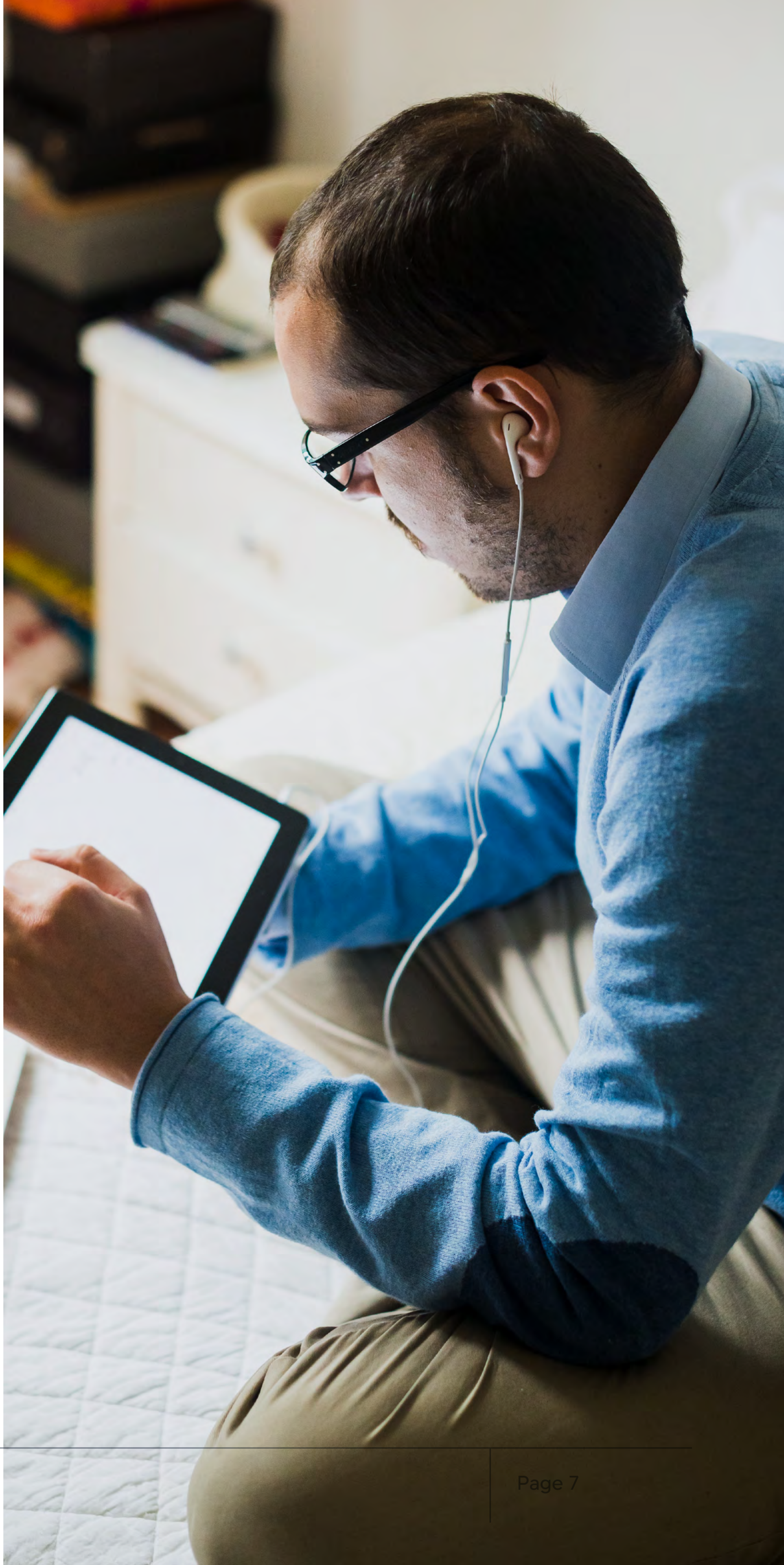
1. Use JavaScript comments to embed the configuration information directly into the file.

2. Use a JavaScript, JSON, or YAML file to specify the configuration for the directory and its subdirectories. This can be in the form of an .eslintrc file, or as an eslintConfig field in the package.json file. ESLint will scan for and read the underlying file, or you can specify the configuration file on the command line.

The following example code is from a JSON file configured for ESLint from an .eslintrc file.



```
  "devDependencies": {
    "eslint": "3.5.0"
  },
  "scripts": {
    "start": "NODE_ENV=production node app.js",
    "test": "npm run lint && npm run custom-tests && echo 'Done.'",
    "lint": "node ./node_modules/eslint/bin/eslint . --max-warnings=0 && echo '✔  Your code looks good.'",
    "custom-tests": "echo \"(No other custom tests yet.)\" && echo",
    "debug": "node debug app.js",
    "deploy": "echo '' && echo '' && echo 'Let us see if you are ready to deploy...' && echo '--' && git status && echo '' && e
  },
```



Best Practices for Error Handling

Good error handling, which maximizes an application's ability to recover gracefully from unexpected errors, can make a big difference in an application's reliability and especially its user experience. Error handling can also make it easier and faster to fix recurring errors in a production application.

The following best practices are taken from an excellent article on Node.js error handling:

- Given functions should deliver operational errors either synchronously (using *throw*) or asynchronously (with a callback or event emitter), but not both.
- When writing new functions, clearly document the arguments for it, the types, and any constraints (e.g., "Must be a valid IP address"). This documentation should also include any operational errors that may occur, and how those errors are delivered.
- Missing or invalid arguments are programmer errors, and you should always use *throw* when that occurs.
- Use the standard Error class and its related properties when delivering errors. Add as much useful information in separate properties.

Node.js Specific Error Handling Use Cases

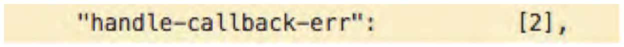
Reporting Async Errors

Errors on asynchronous APIs can be reported in multiple ways:

- Most methods that accept a callback function will accept an Error object passed as the first argument on a function.
- Errors can be routed to an EventEmitter object's error event when an asynchronous event is called.
- There are some asynchronous methods in the Node.js API that may still use the *throw* method that must be handled using *try/catch*.

Handling Errors From Node.js Callbacks

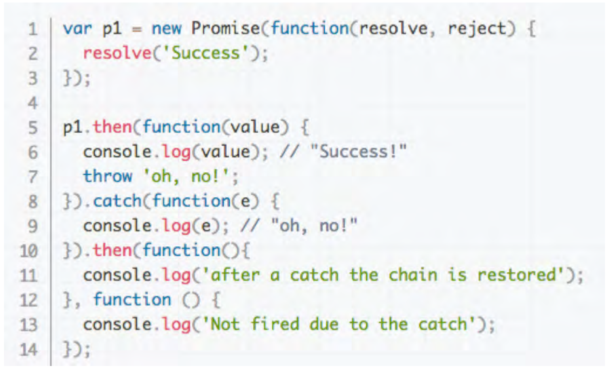
One of the critical errors in Node.js is from callbacks. Here is a sample of how the code should look in the Rules section of the .eslintrc file:



```
    "handle-callback-err":            [2],
```

Handling Errors From Promises/Catch

The catch method assists error handling during the composition of the promise. An example is provided below:



```
var p1 = new Promise(function(resolve, reject) {
  resolve('Success');
});

p1.then(function(value) {
  console.log(value); // "Success!"
  throw 'oh, no!';
}).catch(function(e) {
  console.log(e); // "oh, no!"
}).then(function(){
  console.log('after a catch the chain is restored');
}, function () {
  console.log('Not fired due to the catch');
});
```

Node.js Instance Management

Finally, remember that Node.js errors will often take down the whole service instance, so you need to deal effectively with this level of failure.

- Use a process supervisor like forever or supervisor to watch your Node.js application. If an instance of the app is in an error state, they'll restart it instantly, resulting in minimal downtime.
- Use Node.js' built-in cluster module to "spawn" multiple versions of the same app that listen in on the same socket. The "cluster master" creates new instances and will also restart failed instances.

Confirming Your Code Meets Security Requirements

Security is obviously a very important, and potentially very complex topic. We won't attempt to cover every possible angle here, so we encourage you to seek additional security references.

As a rule, however, every Node.js application should address the following security considerations and threat categories before it enters production:

- Clickjacking: Refers to a category of attacks that tricks users into launching unintended events in the user interface
- Content Security Policy: Instructs the client browser which location and related resources are allowed to be loaded in the browser
- CORS: Cross-Origin Resource Sharing
- CSRF: Cross-Site Request Forgery
- DOS: Denial of Service attacks
- P3P: Platform for Privacy Preferences
- Socket Hijacking
- Strict Transport Security
- XSS: Cross-Site scripting

TLS/SSL

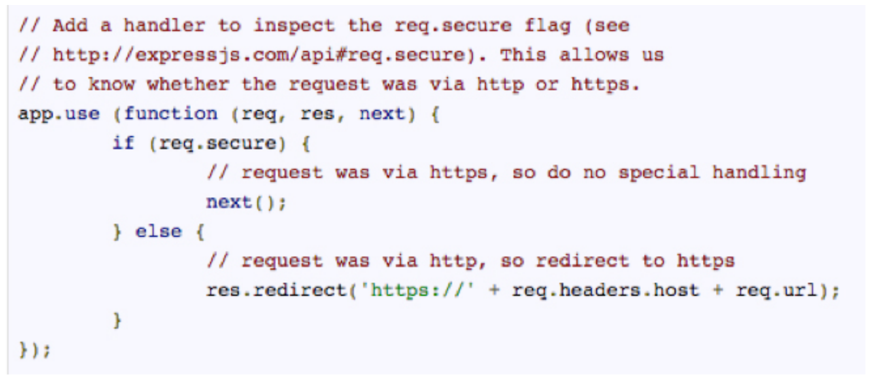
You should ensure that any cookies sent in your application are secure in this layer. The *secure* attribute of cookies instructs the browser to only send that cookie if it is being sent by HTTPS.

TrustProxy Setting

If you are running your application on a proxy server, you will need to use the *TrustProxy* setting if you need to obtain the IP address of the requesting client browser. You can set the *TrustProxy* setting to one of the following types:

- Boolean
- IP addresses
- Number
- Function

Enabling *TrustProxy* also turns on reverse proxy support to assist in redirecting traffic to HTTPS instead of using HTTP protocol. The *req.secure* variable can also help you in sending traffic via the HTTPS protocol as well. The following is an example of how you can use the *req.secure* variable to accomplish this:



```
// Add a handler to inspect the req.secure flag (see
// http://expressjs.com/api#req.secure). This allows us
// to know whether the request was via http or https.
app.use (function (req, res, next) {
        if (req.secure) {
                // request was via https, so do no special handling
                next();
        } else {
                // request was via http, so redirect to https
                res.redirect('https://' + req.headers.host + req.url);
        }
});
```

Configuring for a Production Environment

It's important that your application environment is correctly configured for production release. This is often very different to the configuration used for testing and staging environments and is typically controlled through the use of environment variables.

Heroku is a good tool to create and manage environment variables for Node.js applications. You will need to configure the Node Package Manager (npm) within Heroku before setting these variables. npm can read any configured environment variables, so long as these variables begin with *NPM_CONFIG*.

Node.js Environment Variables

Production API Keys and Credentials
The use of environment variables will allow you to store API keys and related credentials, rather than having to assign a global variable for them. In Node.js, you can access the environment variables through the *process.env* property.

Setting the Production Node
To ensure that the application knows it is running in a production environment, you must set *NODE_ENV=production*. This ensures that the application pulls the production configuration when running in your production environment.

Process Clustering
You may want to take advantage of the cluster module to launch multiple threads to handle the load of several Node.js processes. To set up process clustering, be sure to include the cluster variable as follows:
var cluster = require('cluster);.

Review Hosting Requirements

Before deciding on a hosting provider for your application, be sure to review your config/env/production.js file to account for any idiosyncrasies related to the provider.

Here are a few providers available to host your Node.js application:

- Heroku
- Microsoft Azure
- Google Cloud platform
- Digital Ocean
- Amazon Web Services

Review Load Balancing Options

The Node.js cluster module can be used to enable application load balancing. There is also a "sticky session" module in Node.js that takes incoming connections to the application and routes them based on the originating IP address.

Database Configuration

Production Session Store

Session stores are used for multiple databases in Node.js. This allows your application to keep a user session intact if the application has to connect to multiple databases during that same session. Redis is an open-source,
in-memory store used as a database that can handle session stores.

This following example demonstrates how Node.js can connect to Redis:

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

var express = require('express');
var app = express();

app.use(express.session({
    store: sessionstore.createSessionStore({
        type: 'redis',
        host: 'localhost',          // optional
        port: 6379,                 // optional
        prefix: 'sess',             // optional
        timeout: 10000              // optional
    })
}));
```

Best Practices with Redis

There are occasions where Redis may timeout if there are lengthy periods of inactivity. In order to avoid that, we recommend writing a PING command through a timer command. The following example can help to avoid this issue:

```
setInterval(function(){
        console.log('redisClient => Sending Ping...');
        redisClient.ping();
}, 60000); // 60 seconds
```

Queues and Durability

For queues, we recommend Redis Simple Message Queue, as it's a lightweight message queue for Node.js applications. It requires no dedicated server, just a Redis server.

NodeChef

NodeChef provides a platform to manage and scale Node.js with a number of popular SQL / NoSQL databases including Redis, MongoDB, PostgreSQL, and MySQL. It's worth considering to manage the datastore for your Node.js applications.

Infrastructure Checklist

Finally, remember to have the following items appropriately configured for your production application deployment:

1. Load balancer
2. Database
3. File system for file uploads
4. Session store (if applicable)
5. A PubSub queue for scaling WebSockets
6. A job runner if you are scheduling jobs for your application

It's important to note that if you need to scale up different endpoints in your application, you can still have all of the same servers hitting the same codebase. If necessary, you can configure different installations of your codebase and disable or enable the routes based on that configuration.

Deployment Considerations

It's important to not overlook a number of issues associated with moving your Node.js application successfully across a staging environment and into production. This includes some operational considerations that are specific to Node.js applications.

Dealing successfully with these issues can ensure a much faster and more efficient deployment process — as well as much better relationships with other team members involved in the process.

Deployment Options

There are several options for deploying your Node.js application into production. The following recommendations (courtesy of StrongLoop) should help you decide which one is most suitable for your production environment:

- Package dependencies and deploy using Git branches or npm
- Deploy and run using Node.js supervisors or managers

Package Dependencies and Deploy Using Git Branches or npm
If using npm, you should be aware that there are operating system-level dependencies for some npm packages. Some native compilation may also be required for these packages.

For private npm packages, you will need an npm credential on the server(s) where the npm install command will need to run. This credential will give permission for private packages to be installed.

Deploy and Run Using Node.js Supervisors or Managers
As discussed in the section on error handling, use Node.js supervisors and managers to automatically detect and handle failures.

Deployment Failures

No matter how carefully you and your team prepare to deploy a Node.js application, experienced developers know that a deployment failure can sometimes still happen.

Some potential failures are outliers — events that are unlikely to occur in a real-world production environment. Others are fairly commonplace. Either way, it's still useful and important to simulate these failures and assess how your application responds to them.

Some examples of these types of failures and the resulting Node.js responses include:

- Processes crashing due to unhandled exception errors.
- RAM overflow: This typically occurs when processes run out of RAM. However, this can also occur if a Node.js process gets overwhelmed.
- Memory leaks, with or without an overflow.
- Runaway recursive function: A message will state "Maximum call stack size exceeded."
- CPU lockups: A process is locked up because the CPU is overwhelmed or caught in an unending blocking loop. The culprit of this loop can usually be found in a *'while(true) {}'* statement in the code.
- No server responses: This occurs if you forget to call *'res.send()'* in the code. The default TCP timeout in Node.js applications is 120 seconds.

All of these scenarios can be tested using the sails-hook-dev tool. It provides diagnostic and debugging information during development. You can use this tool to check the following:

- Memory usage or environment variables of the currently running Node.js process
- Configuration
- Installed versions of dependencies
- Fetch session data for the current user
- Force Node.js' garbage collector to run

Now that we've covered the fundamentals of an effective pre-production workflow, we'll assume that you've pushed your Node.js application successfully into production. In the next chapter, we'll look at what to expect and how to respond during the critical first 24 hours after deployment.

Chapter 2
Staying the Course: The First 24 Hours of Your Node.js Deployment

No matter how you see it, deploying an enterprise application can be harrowing. According to one recent survey, 30% of all application deployments fail. Another survey reported that 77% of organizations have software production release problems. Clearly, anyone tasked with deploying an application should be ready for things to go wrong — perhaps badly wrong.

We'd like to tell you that the Node.js deployment process will be less challenging than these industry norms. And certainly, a robust pre-production process can help to minimize the impact of bugs, configuration failures, and other avoidable problems.

Even the best preparation, however, can't keep Murphy's Law from governing your Node.js deployments: *Anything that can go wrong with a production release will go wrong*. Many of those problems — avoidable or not — will surface during the first 24 hours after an application enters production use.

In this chapter, we'll discuss some of the most common examples of these "Day One" deployment problems — particularly those that result in crashes or other high-impact issues. We'll also touch on the use of application monitoring to help you detect and manage problems more effectively.

As this chapter's title suggests, it's important to have patience and stay the course during this time, work through problems as they appear, and pay special attention to our advice (Rule One) about sticking to a process for pushing out fixes.

Two Keys to Surviving Day One — and Beyond

Before we dive into specifics, we want to spotlight two very important pieces of advice. They'll make the difference, in many cases, between a challenging but successful deployment process, and a nightmare scenario where routine problems spiral into crises.

1. Rule One for Production Emergencies: Stick to Your Process Guns

Application crashes and other visible, potentially high-impact failures can trigger a panic response. Fixing the problem is the only thing that matters.

This is a recipe for disaster. The solution is always — *always* — to follow your established protocol for pushing changes to production.

Keep in mind that "simple" changes often become *less* simple when you have a chance to think about them. In a clustered environment, for example, making a direct change will force you to mimic the exact same change on every server. Even a random missed keystroke can turn into a situation much worse than the original problem.

There's no mistake worse than one that's both self-inflicted and avoidable. Stick to a process that ends with a solution — not a crisis.

2. Application Monitoring: Your New Best Friend

It's hard to eliminate unpleasant surprises completely; they come with the territory when you build business applications. What you can do is minimize the gap between when a problem occurs —or even when warning signs of a problem appear — and when you learn about them.

A good application monitoring package gives you this capability, ensuring that you learn about a problem with an instant notification instead of messages from upset users (or your boss).

Many monitoring solutions are available that work well with Node.js applications — noteworthy examples include AppDynamics' Unified Monitoring. Include monitoring in your standard deployment process, and ensure it's in place during the critical first 24 hours in production.

Common Day One Application Issues

Let's shift gears now from general advice to some specific issues you're likely to encounter during a Day One deployment window. As always, we'll start with a reminder that the following list isn't meant to cover every possibility, but it will cover the most common post-deployment problems and a sense of the kinds of issues you'll encounter here.

1. Crashing Node Processes

Node.js is architected to run on a single process. This is one of its defining qualities — enabling applications that are lightweight yet highly scalable. One of the trade-offs of this approach, however, is that when an issue crashes the process — uncaught exceptions are the most common reason why this happens — it essentially crashes the entire application.

While you're chasing a crashing Node.js process, however, remember that the problem may also involve the platform being used to host your application. That's why we recommend reviewing appropriate, platform-specific guidelines for troubleshooting Node.js applications — for example, Microsoft Azure Web Services and the Heroku Platform as a Service (PaaS) environments.

1A. Fixing Node.js Processes

Once again, the best way to deal with Node.js process crashes is to use Node.js supervisors and managers to automatically detect and handle failures.

2. Exceeding API Rate Limits

Rate limiting is a common practice among web developers, since it gives them a very effective way to restrict a variety of activities within defined parameters. Rate limiting can be used to restrict the number of user queries, as an event-throttling tactic, or as a way to limit API requests.

2A. Defining Application Rate Limits

Rate limits are typically defined by user session, rather than as global limits that apply to the application as a whole. They can take the form of temporal limits (e.g., the number of requests within a 15-minute window) or as limits on specific requests (e.g., GET and POST requests).

Deploying an application can be a moment of truth for a developer's use of rate limiting in a Node.js application. The actual traffic your application receives can differ considerably from predicted traffic — and once an API rate limit has been exceeded, the application will essentially freeze.

2B. Rate Limit Workarounds

There are a few fairly simple ways to adapt an application to any API rate limit issues you encounter in a production application:

- **Caching:** This involves inserting the raw, JSON-encoded data into the database and provides the following benefits:
  - Gives active users higher priority.
  - Adapts to related search results, responding wherever possible, directly from cache and thereby reducing the number of API calls required.
  - Clever design can also enhance the benefits of implementing caching. For example, If you know that there is a hard limit to the maximum number of responses from a given search string and the data footprint is not excessive, why not pre-fetch the full set of responses? That way you can reduce the number of API calls even further by having all possible responses already sitting in cache.

Keep in mind that while implementing a caching solution is almost always beneficial to application performance, there is a trade-off in the form of increased memory usage. Even if you attempt to avoid this, for example, by using a cursor/pagination/map-reduce approach to access pages of data at a time, this just leads to more requests — which means you're getting closer to your rate limit.

If caching doesn't ultimately provide the benefits you were expecting, then do what's necessary to keep your application available and as functional as possible — which means, in this case, temporarily disabling the feature(s) causing the problem and contacting the provider for help with a permanent solution.

3. Troubleshooting WebSocket Issues

At this point, most developers working with Node.js will be acquainted with a WebSocket and with the benefits it offers versus using HTTP. As of this writing, the most popular WebSocket library for Node.js is Socket.IO, which also benefits from being relatively easy to implement and use.

Besides Socket.IO, there are many different WebSocket libraries available for Node.js today. While each has its own API, they're all built on top of TCP and do basically the same things.

No matter which WebSocket library you use, any problem that impedes communication between your server and clients is obviously an urgent one. The following procedure gives you a quick and easy way to reproduce a suspected WebSocket issue:

1. Create a WebSocket server in its own Node.js process.
2. Connect to this server using the WebSocket client. This client should be running a separate process.
3. End the server process created in the first step.
4. Check the status code. If an error code of `1000` (normal) is reported, then this is incorrect, as the browser would normally report an error code of `1006` in this circumstance.

4. Dependency Issues

These issues will typically manifest prior to going live in production, or when updates are made to your application in subsequent pushes to production. One typical issue related to dependency errors is when the message "MODULE_NOT_FOUND" is displayed. This error will occur when a given path in the application does not exist.

5. File Upload Issues

The server can be overwhelmed by upload requests immediately after deployment to production. This occurs when the files are being uploaded to the local file system. This is more of an operational issue than a bug or other software-related failure, but it's still a significant source of unplanned downtime risk.

If an error occurs during file uploads, however, there's another potential issue to consider. Typically, a body parser such as Skipper can handle file uploads without a hiccup. An error message suggests a disconnect in that process.

6. Denial-of-Service (DoS) Attacks

Denial-of-service attacks pose a complex problem involving multiple layers of protection across the networking stack. There isn't much that can be done at the API layer, but configuring Sails appropriately can mitigate certain types of DoS attacks:

- Sails sessions can be configured to use a separate session store (e.g., Redis), so your application can run without relying on the memory state of any one API server. This allows you to distribute load by deploying multiple copies of your Sails app across as many servers as required. You do need to front end your servers with a load balancer, configured to ensure incoming requests are always directed to the least busy server. This significantly reduces the risk of one overloaded server becoming a single point of failure.
- If you are **not** using the long-polling transport enabled in sails.config.sockets, then Socket.IO connections can be configured to use a separate socket store (e.g., Redis). This eliminates the need for sticky sessions at the load balancer and the possibility of would-be attackers directing their attacks against a specific server.

Looking Ahead: Time to Hit the Open Road

The good news about Day One surprises (and there will be surprises) is that you're going to learn a lot about building better Node.js applications and about launching your applications with fewer post-deployment issues. While problems can and will continue to happen in the future, truly serious problems will probably be fewer and farther between.

Even better news is that you're also over the hump with your current Node.js deployment. From here, you'll be dealing with a more stable and reliable application — and that, in turn, frees you to focus on ways to improve your application's performance and to upgrade your own process for building, testing, and deploying Node.js applications.

For the first time since beginning this process, you'll probably feel a real sense of momentum and progress — getting your Node.js project out of traffic and onto the open road. Let's see what this means in terms of your next-step projects and priorities — our topic for Chapter Three.

Chapter 3
Ongoing Management

Having successfully deployed, this final chapter looks at the ongoing management of your Node.js application. This isn't too different from any other application rollout, but there are a couple of specifics you should watch out for:

Memory Leaks

Memory leaks are a common problem area for all Java-based application servers. If you suspect a memory leak in your application, first check that you're using recommended production settings, then check the following:

- Check the application-level code in an effort to isolate the memory leak to a single endpoint.
- A common memory leak in Node.js applications is when you are using promises in the code but forget to include a catch() statement.
- If the increasing memory usage cannot be pinpointed to a specific endpoint or anything else within the application code, then try to reproduce the leak in a separate new application.
- This new application should run in streamlined mode without any add-ons and with recommended production settings. If an issue is found, create an example repository that recreates the memory leak.

Tuning the Garbage Collector

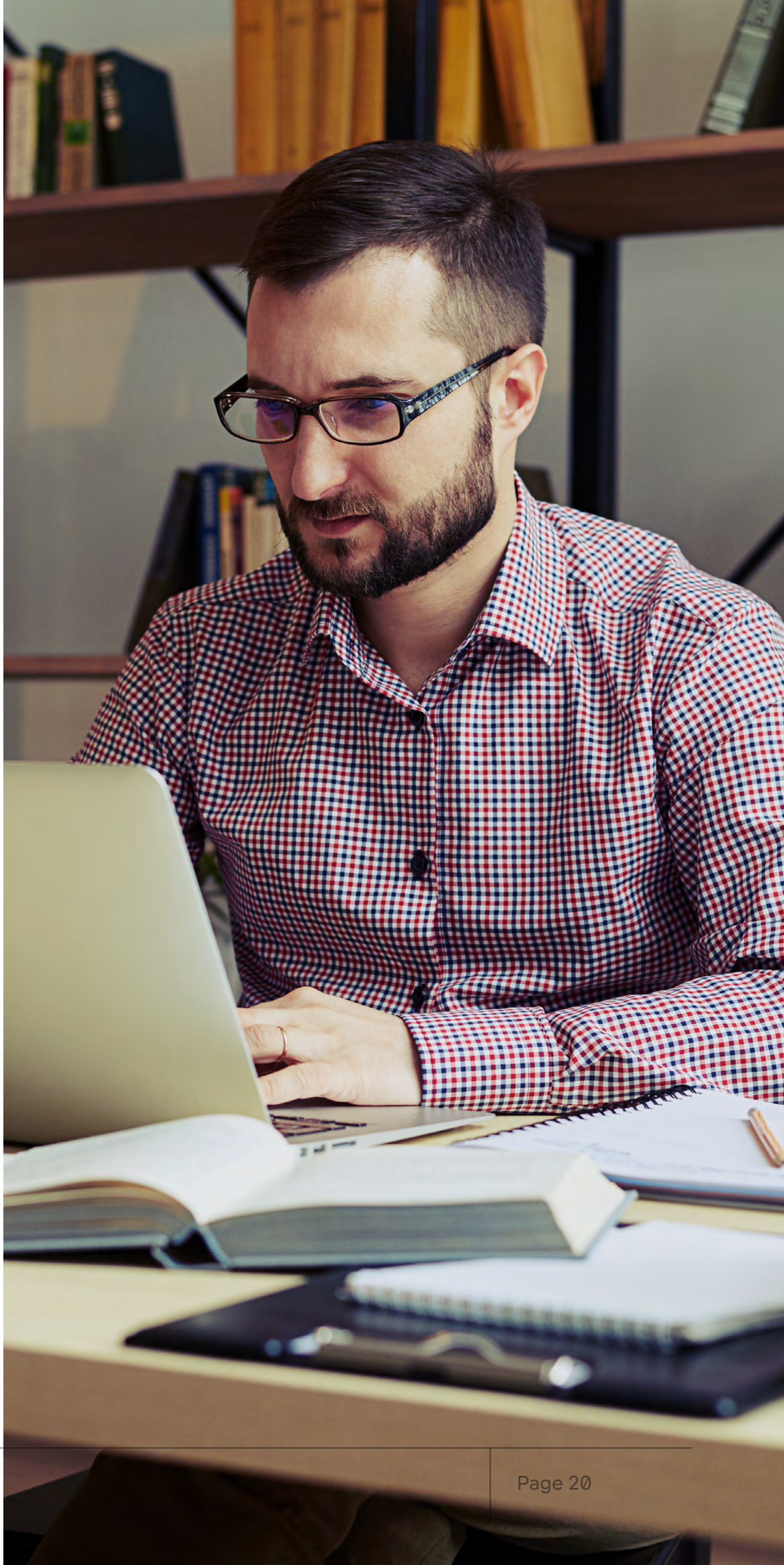
Node.js sets a limit of 1.5 GB for long-lived objects by default. If this exceeds the memory available, Node.js could allow your application to start paging memory to disk.

To gain more control over your application's garbage collector, you can provide flags to the underlying JavaScript engine in your profile:

```
web: node --optimize_for_size --max_old_space_size=920 server.js
```

If you'd like to tailor Node.js to a 512 MB container, try:

```
web: node --optimize_for_size --max_old_space_size=460 server.js
```



Managing Node.js Concurrency

Remember that Node.js has a limited ability to scale to different container sizes. It's single-threaded, so it can't automatically take advantage of additional CPU cores. Furthermore, it's based on V8, which has a memory limit of approximately 1.5 GB, so it cannot automatically take advantage of additional memory.

Node.js apps must fork multiple processes to maximize their available resources, which is referred to as "clustering." This is supported by the Node.js Cluster API which you can invoke directly in your app. With the Cluster API, you can optimize your app's performance and the Heroku Node.js buildpack provides environment variables to help.

Tuning the Concurrency Level

Each app has unique memory, CPU, and I/O requirements. The Heroku buildpack provides reasonable defaults through two environment variables: WEB_MEMORY and WEB_CONCURRENCY. You can override both to fit your application.

- WEB_MEMORY specifies, in MB, the expected memory requirements of your application's processes. It defaults to 512.
- WEB_CONCURRENCY specifies the recommended number of processes to cluster for your application. It's essentially MEMORY_AVAILABLE / WEB_MEMORY.

If you need more information, then check out the Heroku Dev Center, which is a great source of articles on tuning and managing your Node.js app, including how to manage Node.js concurrency.

Monitoring

As a final thought, don't ignore monitoring, which is vital to maintain the stability of your application deployment and to detect subtle regressions that may result in application slow-down or outright failure if left unchecked.

An APM solution like AppDynamics provides end-to-end insight into application behavior and provides specific monitoring capabilities for the Node.js stack.

To Conclude

Hopefully, this short eBook has provided helpful insight into a best practice approach to deploying and managing your Node.js applications. Like containers and microservices, Node.js best practices continue to evolve, but you're already off to a great start and well-placed for success in the rest of your Node.js journey.