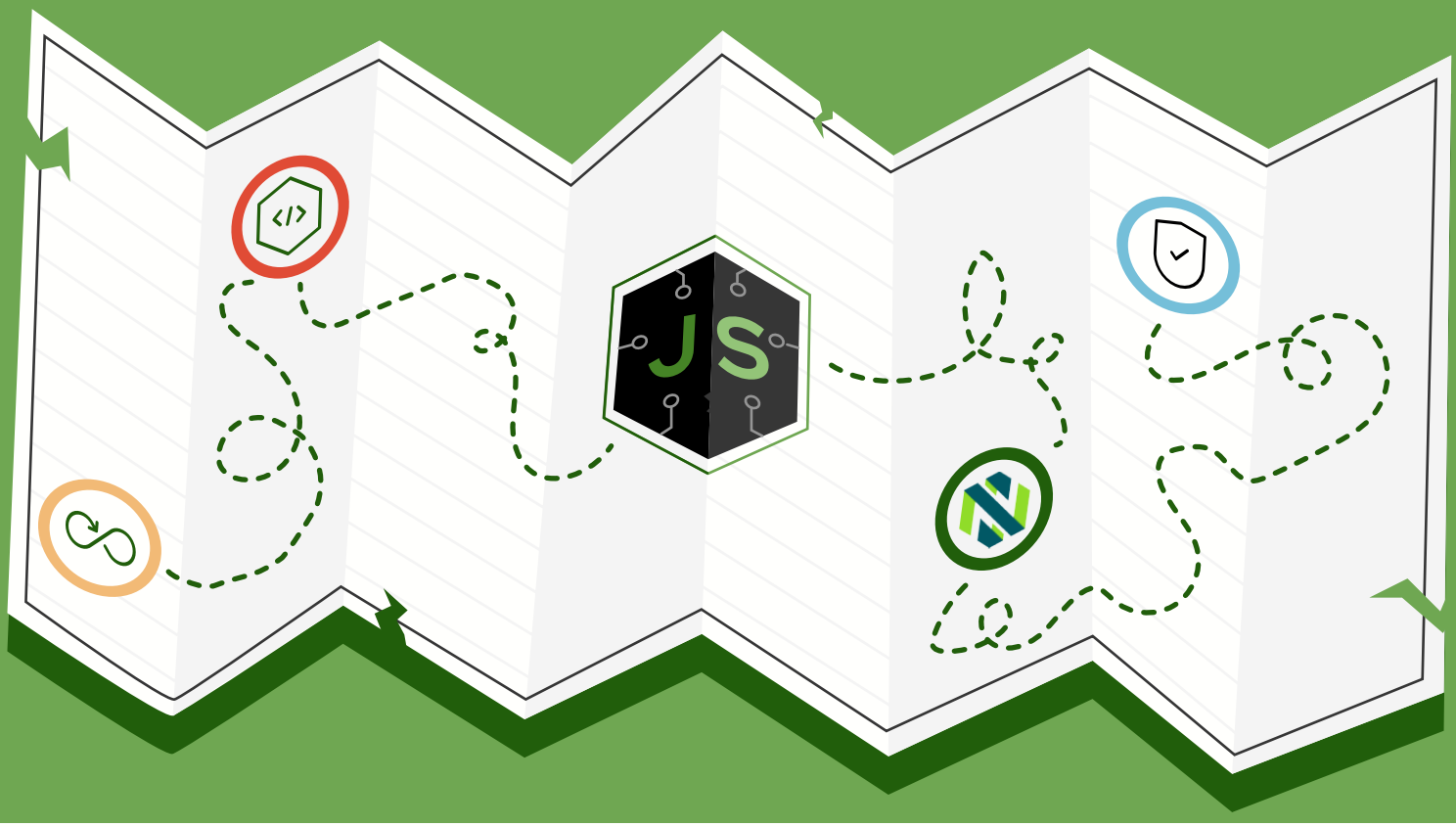
Published by Red Hat Developer

A Developer's Guide to the Node.js Reference Architecture



Curated by Michael Dawson and Luke Holmquist

Foreword by Matteo Collina

A Developer's Guide to the Node.js Reference Architecture

Curated by Michael Dawson and Luke Holmquist

Red Hat Developer

Table of Contents

Foreword

My journey with Node.js began in 2010, when it was still in its infancy. I was looking for a fast runtime that could handle tens of thousands of concurrent connections on a single core, and that's when I discovered Node.js. Over the past 14 years, I've had the privilege of witnessing its transformative power, not just as a tool but as a vibrant ecosystem and community that continuously pushes the boundaries of what's possible in software development.

When Node.js first appeared on the scene, it revolutionized how we think about server-side JavaScript. For the first time, JavaScript developers could leverage their skills on the server, using a single language for both client and server-side programming. Not only did this unification streamline development processes, but it also unleashed a new wave of creativity and efficiency, as developers could now build full-stack applications with unprecedented ease.

However, Node.js's true strength lies in its community. An open source project of this magnitude thrives on the contributions, collaboration, and shared vision of developers from around the globe. The Node.js ecosystem, powered by npm, offers an extensive library of modules and packages that extend its functionality and simplify complex tasks. This collaborative spirit fosters a dynamic environment where innovation flourishes and challenges are met with collective ingenuity.

In truth, the Node.js/npm combo solved one of the greatest challenges of software development: reusing software at scale. So far, the community has created over 3 million packages that can be downloaded and reused to build greater things. This is why having a reference architecture for Node.js is important and makes this book special.

This book is a testament to Node.js's enduring appeal and versatility and a guide to navigating the infinite possibilities of the npm registry while assembling a production-ready Node.js application.

As a member of the Node.js Technical Steering Committee, I have worked with Michael and his team for years to improve Node.js. I can fully testify that the methodology they used to assemble the reference architecture presented in this e-book is solid. While I don't endorse all of the opinions in this book, it's important for a company to think through these aspects of Node.js development and capture their shared experience.

As you embark on this journey, I urge you to not just be a passive reader but to immerse yourself in the Node.js community. Engage in discussions, contribute to projects, and share your experiences. The strength of Node.js is amplified by the collective effort of its community members, and your active involvement will enhance your skills and contribute to the growth and evolution of this remarkable platform.

Node.js is not just a technology; it's a movement that embodies the spirit of collaboration, innovation, and relentless pursuit of excellence. I am thrilled for you to explore the transformative possibilities that Node.js offers and to become a part of its vibrant community. Together, we can continue to push the boundaries of what is possible and shape the future of web development.

—Matteo Collina
Node.js Technical Steering Committee Member
OpenJS Board Member
Platformatic Co-founder & CTO
May 2024

Preface

Michael Dawson and Luke Holmquist

This book introduces the Node.js reference architecture from Red Hat and IBM. First, we'll explain the reasoning behind developing the Node.js reference architecture—both what we hope the architecture will offer our developer community and what we do not intend it to do. Each chapter in this book provides a detailed look at different sections of the reference architecture.

Before we dive in, it's important to acknowledge that the Node.js reference architecture will always be a work in progress. The development team regularly works through different areas, discussing what we've learned and distilling that information into concise recommendations and guidance. Given the fast pace of development in the JavaScript ecosystem, the reference architecture might never be "finished." Instead, we'll continue updating it to reflect what we learn through new Node.js production deployments and ongoing experience with our deployments at scale. The reference architecture is meant to reflect our current experience and thinking, which will evolve over time.

Why we need a Node.js reference architecture

The JavaScript ecosystem is fast-moving and vibrant. You only need to look at the growth rate of Node Package Manager (npm) modules to see that. In 2016, there were approximately 250,000 npm packages. In 2018, that number climbed to around 525,000, and in 2020 it was roughly 1.1 million. These numbers represent considerable choice and variety in the JavaScript ecosystem. That is clearly a strength for flourishing innovation and testing new ideas.

On the flip side, the wide variety of options can make choosing among Node.js packages very difficult. For any module, you might find several equally good choices, as well as several potentially very bad choices. Every application has a "secret sauce" that is key to its success. It is imperative to find the best fitting, newest, or most innovative package to use for this area of the application. For the rest of the application, you likely want something that works and for which you can share any experiences or best practices across your organization. In the latter case, having a reference architecture can help teams avoid relearning the same things again and again.

What the reference architecture is

Our Node.js teams at Red Hat and IBM can't be experts on all of the large number of JavaScript packages in the npm registry. Likewise, we can't be involved in all of the projects to the level that we are involved in the Node.js project. Instead, our experience is based on our broad usage of Node.js. This includes large-scale deployments like the Weather Company, as well as the work that our consulting groups do with customers.

If every internal team and customer who asks for help with their Node.js application uses different packages, it will be much harder to help them. The question is, how do we share our knowledge across the organization?

We want to help our internal teams and customers make good choices and deployment decisions. In cases where a team doesn't need to use a specific package, we can recommend a package based on the experience we've built across Red Hat and IBM. As developers, we can use the Node.js reference architecture to share and collaborate across teams and projects and establish common ground within our deployments.

What the reference architecture is not

We have described what we hope to do with the Node.js reference architecture. It is just as important to be clear about what we are not trying to do.

First, the reference architecture is not an attempt to convince or force developers to use the packages we choose. Deployments are varied, and there will be good reasons to use specific modules in different circumstances.

Second, we do not claim that our recommendations are better than the alternatives. As noted, you will often find several equally good packages or approaches available in the JavaScript ecosystem. Our recommendations favor what the Red Hat and IBM teams have

used successfully and the technologies we are familiar with. We are not attempting to steer anyone to the “best” choice, but rather to a “good” choice. Having a reference architecture maximizes the likelihood of leveraging lessons already learned and having common ground so that we can help each other.

Why we wrote this book

The Node.js development team is having interesting discussions as we work through each section of the reference architecture. At the same time, we are trying to keep the reference architecture's content concise and to the point. As we've mentioned, the goal is to provide good choices for the application's general architecture so that developers can focus on the application's "secret sauce." In most cases, developers using the reference architecture will want to know what package or technology to use and how. As a result, the reference architecture won't include much about the interesting background and discussions that led to our decisions.

This book will share the viewpoints gained from our internal discussions. We think you’ll find the varied experience of developers across the Node.js team gets you thinking. We learned something from every section we went through, and we hope you will, too.

How to get started

This book include 16 chapters as follows:

- Chapter 1: Logging
- Chapter 2: Code consistency
- Chapter 3: GraphQL
- Chapter 4: Building good containers
- Chapter 5: Choosing web frameworks
- Chapter 6: Code coverage
- Chapter 7: TypeScript
- Chapter 8: Securing Node.js applications
- Chapter 9: Accessibility
- Chapter 10: Typical development workflows

- Chapter 11: npm development
- Chapter 12: Problem determination
- Chapter 13: Testing
- Chapter 14: Transaction handling
- Chapter 15: Load balancing, threading, and scaling
- Chapter 16: CI/CD best practices

Each chapter is self-contained so you can read the book in the sequence provided or choose to dive into topics that are of particular interest. Each section will take you about 5 to 10 minutes to read through.

Red Hat Developer

Building and delivering modern, innovative applications and services is more complicated and fast-moving than ever. Red Hat Developer has the technical tools and expertise to help you succeed.

The Red Hat Developer program provides many member benefits, including a no-cost subscription for individuals and access to products like Red Hat Enterprise Linux and Red Hat OpenShift. Learn more.

Chapter 1: Logging

Michael Dawson and James Lindeman

In this chapter, we'll dig into logging in Node.js. Understanding what tools to use for logging in your Node.js applications and deployments is a critical step to ensuring they run well in production.

First, we will discuss why the Node.js reference architecture recommends Pino as a great tool for logging in to your Node.js application. Then we'll walk through an example based on an IBM team's experience with integrating logging into a Node.js application.

> **Note:** As with all our Node.js reference architecture recommendations, we focus on defining a set of good and reliable default choices. Your team might deviate from some of these recommendations based on what best fits your use case.

Real-world implementation of logging guidance

How do all these recommendations translate to a real-world implementation? While the reference architecture sticks to concise recommendations, we want to expand a bit on the "why" behind our decision and offer real-world examples of Node.js usage key concepts. This section discusses one team's implementation of these logging principles in detail.

At this point, Jim will share details about his team's experience with integrating logging into a Node.js application at IBM. Hopefully, you'll see how this perspective (along with others) has shaped our guidance and preceding recommendation.

Structured logging

A good logging strategy ensures that your logs are structured as carriage-return separated JSON blocks; this makes it easy for modern log repositories and viewers like LogDNA, Kibana, and Loggly to sort and filter various keywords in the JSON block. Even though modern logging repositories will turn string-based logs into JSON logs internally, providing JSON logs allows your applications to define their own custom keywords to sort or search. It's still common for shell scripts at the startup of a container to not use JSON logs but to have a Node.js process in the same container use JSON logging after startup.

Development teams sometimes push back against this, because logs become harder to read when viewed in a log file. A workaround we use is to use a log formatting method to sort specific fields to come in the same order as the non-JSON format. For example, first in the serialized JSON block is the timestamp, level, then message, with the rest of the keywords following them in any order.

Modern logging repositories handle the storage, filtering, and visualization of the logs that your application generates. Figure 1-1 shows an example of viewing logs that the IBM Log Analysis service has fed into LogDNA. (The `graphql_call` and `transaction_id` fields you can see here will be discussed in the Logging formatters and thread-local variables section on page 3.)

Figure 1-1: Viewing logs in LogDNA.

Static and dynamic log-level management

Early on, you should define a standard set of log levels that application developers should use. These might differ based on your chosen logging tool, but having them well defined helps ensure consistent use across your team.

Your team should review all the spots an application will generate an error log to confirm that the different variables needed to debug that error path are logged. Doing so helps you avoid needing to re-create the problem later with more detailed logging. Your goal should be "First Failure Data Capture" so that you don't have to recreate problems to debug customer issues. If you need additional data to debug a customer issue, you should create a separate issue to track adding additional logging to cover the additional data that was needed in that error path.

As noted, the initial application logging level is determined by the environment variable setting. However, we found certain "corrupted state" cases where we wanted to change the logging level to be more verbose without redeploying or restarting the application. This occurred when we started to have occasional database or queuing-service errors. We used a hidden REST endpoint that was not exposed publicly, but we could still invoke it using admin-routing tools like `kubectl proxy`. The REST endpoint could communicate with the logging library to change the logging level dynamically.

Logging formatters and thread-local variables

Logging formatters are customizable methods that allow applications to add or transform data into the outputted JSON logs. Using formatters to export specific thread-local variables into the logs is a powerful technique for getting useful data that your support teams will want in all your logs.

For example, your support team might want the ability to filter logs to a specific failing REST call. Generating and storing a REST call's `trace-guid` as a specific thread-local variable early in the REST processing (i.e., one of the first express middleware layers) allows anyone following logs in that REST call to then share that same `trace-guid`.

Customer error messages that provide that `trace-guid` make it simple for your support team to help the customer by filtering that `trace-guid` to know exactly what the customer was doing. Otherwise, your support team has to hunt through logs for errors with similar timestamps, which is time-consuming. There are other ways to get cross-transaction

information, including distributed tracing, but it's often still useful to include this kind of information in logs.

While Node.js doesn't have obvious thread-local variable support like Python and Ruby, it does have that functionality exposed through its `async-hooks` library. Specifically, in the latest Node.js versions, you can access AsyncLocalStorage. Otherwise, if you are still using older Node.js versions, you can use the `cls-hooked` library even back to Node.js 8 to get equivalent functionality.

Good thread-local variables to set in a common authentication-validation layer include `user-guid` (as opposed to an email address for GDPR compliance), `organization-id`, and if the user is an application admin. Those allow security auditors to trace what actions a specific user or organization has done and validate that your own admins have not taken inappropriate actions.

If your application uses GraphQL, you will notice that because all the GraphQL calls share the same entry endpoint, traditional REST trace logs are not very helpful, as they won't indicate which GraphQL call was invoked. We updated our GraphQL entry methods to set the method name in a thread-local variable so that we could include the name in the REST trace logs.

Logging the real IP

In most Kubernetes-based deployments, the source IP address that the applications see is just that of the ingress router and not from the caller. If you use a security firewall in front of your Kubernetes cluster (like this Cloudflare service offered in the IBM Cloud), the caller's IP address will be stored in an `X-Forwarded-For` header. The application endpoint handler can then extract and log the header instead of the REST request's IP address. If you are not using a front end like that, you might have to specifically configure your load balancer and ingress to ensure the public IP is carried into the Kubernetes ingress and then into your application.

Configuration logging at startup

Dumping the application configuration settings at startup can help developers quickly identify and understand configuration problems instead of guessing what the configuration "should be." This provides key information that will save you time investigating issues while limiting the potential for sensitive information in subsequent logs.

However, doing that can sometimes expose values like passwords or keys that should normally remain secret. A common example is when a developer adds a `printenv` at the application's

startup to dump all the environment variables and inadvertently exposes the database passwords also passed in as environment variables.

Node.js libraries like convict make it easy to define how configuration settings are imported. They allow the app to determine if the settings should be imported from mounted config files, environment variables, or both. They also determine what elements in a configuration are optional, what default settings should be, and if a field's value should be treated as sensitive (i.e., value is a password or API key). As long as developers mark the password and API key fields as sensitive in the definition of the configuration, then the configuration object can be safely logged at startup; the sensitive field values will be converted into asterisks.

Because we mentioned sensitive information, it's also important to consider the information your application is gathering and use redaction to remove sensitive fields when logging that information.

Conclusion

We hope this chapter has given you more insight and background on the recommendations and guidance in the logging section of the Node.js reference architecture, along with a deeper dive into some of the team's experience and what we've found useful.

In the next chapter, we'll dive into code consistency in Node.js.

Chapter 2: Code consistency

Luke Holmquist

In this chapter, we will dive into code consistency and how to enforce it with a linter tool like ESLint.

Why code consistency matters

One critical aspect of working on JavaScript projects effectively as a team is having consistency in the formatting of your code. This ensures that when different team members collaborate on the shared codebase, they know what coding patterns to expect, allowing them to work more efficiently. A lack of consistency increases the learning curve for developers and can potentially detract from the main project goal.

When the Node.js teams at Red Hat and IBM started the discussion on code consistency, it quickly became apparent that this is an area where people have strong opinions, and one size does not fit all. It's amazing how much time you can spend talking about the right place for a bracket!

The one thing we could agree on, though, is the importance of using a consistent style within a project and enforcing it through automation.

ESLint

In surveying the tools used across Red Hat and IBM to check and enforce code consistency, ESLint quickly surfaced as the most popular choice. This configurable linter tool analyzes code to identify JavaScript patterns and maintain quality.

While we found that different teams used different code styles, many of them reported that they used ESLint to get the job done. ESLint is an open source project hosted by the OpenJS Foundation, confirming it as a solid choice with open governance. We know we'll always have the opportunity to contribute fixes and get involved with the project.

ESLint comes with many pre-existing code style configurations that you can easily add to your projects. Using one of these shareable configurations has many benefits. By using an existing config, you can avoid reinventing the wheel; someone else has probably already created the configuration you are looking for. Another advantage is that new team members (or open source contributors) might already be familiar with the config you are using, enabling them to get up to speed more quickly.

Here are a few common configurations to help you get started:

- eslint-config-airbnb-standard
- eslint-config-semistandard
- eslint-config-standard
- eslint-config-prettier

A complete list can be found on npmjs.org using this query.

Note we do not recommend any particular code style or ESLint config. It's more important that you choose one standard and that you apply it consistently across your organization. If that is not possible, then you should at least make sure it's used consistently across related projects.

At this point, I must admit we did not really spend that much time talking about where the brackets should go. But that is one of the reasons we suggest looking at one of the existing configs: Adopting existing best practices saves a lot of time (and arguments) so you can spend that time coding instead.

Adding ESLint to your Node.js project

Based on the advice in the reference architecture, the Red Hat Node.js team recently updated the NodeShift project to use ESLint.

Adding ESLint to your project is a pretty straightforward process. In fact, ESLint has a wizard that you can run on the command line interface to get you started. You can run:

```
$ npx eslint --init
```

Then, follow the prompts. I won't go into the specifics of the init wizard here, but you can find more information in the ESLint documentation.

Our team likes using semi-colons, so we decided to use the semistandard config. It was easy to install by running the following command:

```
$ npx install-peerdeps --dev eslint-config-semistandard
```

Then, in our .eslintrc.json file, we made sure to extend semistandard:

```
{
  "extends": "semistandard",
  "rules": {
    "prefer-const": "error",
    "block-scoped-var": "error",
    "prefer-template": "warn",
    "no-unneeded-ternary": "warn",
    "no-use-before-define": [
      "error",
      "nofunc"
    ]
  }
}
```

You will notice that we have some custom rules set up as well. If you have custom rules for your project, this is where you should put them.

Automating the code linter

Having a linter in place is great, but it is only effective if you run it. While you can run the `eslint` command manually to check your code consistency, remembering to run it that way can become burdensome and error-prone. The best approach is to set up some type of automation.

The first step is to create an npm script like `pretest` that will make sure linting happens before your tests are run. That script might look something like this:

```
"scripts": {
    "pretest": "eslint --ignore-path .gitignore ."
}
```

Notice that we are telling ESLint to ignore paths that are contained in our `.gitignore` file, so make sure the `node_modules` folder and other derived files are included in that ignore file. Using an npm script like this easily integrates into most continuous integration (CI) platforms.

Another alternative is to configure hooks so that the linter runs before the code is committed. Libraries like Husky can help with this workflow. Just be sure that these precommit checks don't take too long, or your developers might complain.

Conclusion

It is critical to make sure you enforce consistent code standards across all your projects so that your team can collaborate efficiently. The best way to keep up with that task is to use a linter and automate it as part of your workflow. We recommend ESLint, but you are free to choose whatever tool you want—as long as you have something.

In the next chapter, we'll look at GraphQL in the Node.js ecosystem.

Chapter 3: GraphQL

Wojciech Trocki

In this chapter we dig into some of the discussions the team had when developing the GraphQL section of the reference architecture. You will learn about the principles we considered and gain additional insight into how we developed the current recommendations for using GraphQL in your Node.js applications.

GraphQL in the Node.js ecosystem

GraphQL is a query language specification that includes specific semantics for interaction between the client and server. Implementing a GraphQL server and client typically requires more effort than building REST applications, due to the extensive nature of the language and additional requirements for client-side and server-side developers. To start, let's consider a few of the elements of developing a Node.js application with GraphQL.

Developing a GraphQL schema

When building a GraphQL API, client and server-side teams must define strong contracts in the form of a GraphQL schema. The two teams must also change the way they have been communicating and developing their software. GraphQL internally requires server-side developers to build data-handling methods called resolvers that match the GraphQL schema, which is an internal graph that both teams must build and agree on. Client-side developers typically need to use specialized clients to send GraphQL queries to the back-end server.

Choosing your tools

The GraphQL ecosystem consists of thousands of libraries and solutions that you can find on GitHub, at conferences, and in various forums that offer to resolve all your GraphQL problems. On top of frameworks and libraries, the GraphQL ecosystem offers many out-of-the-box, self-hosted, or even service-based (SaaS) CRUD engines. Create, read, update, and delete (CRUD) engines offer to minimize the amount of server-side development by providing a direct link to the database. We'll come back to this topic later.

Implementing a GraphQL API

When implementing a GraphQL API, we often see a number of side effects on other elements of our back-end infrastructure. A GraphQL API is typically exposed as a single endpoint by our back end, as illustrated in Figure 3-1.

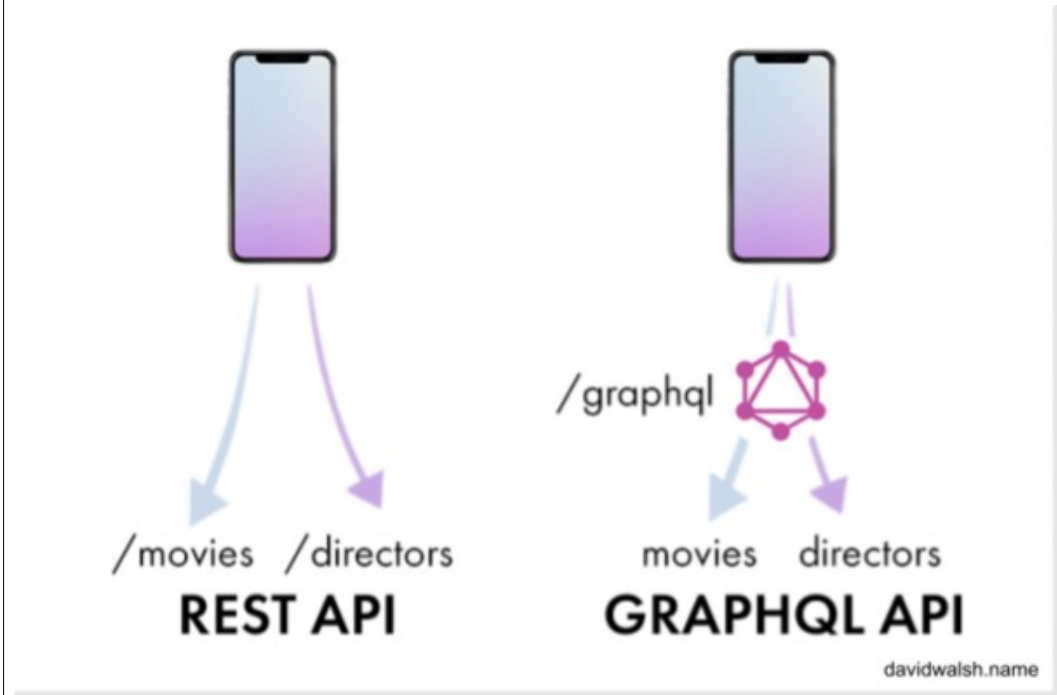


Figure 3-1: Unlike a REST API, a GraphQL API is exposed as a single endpoint.

Adopting the GraphQL API means not only do we need to change the API, but we often have to rethink our entire infrastructure, from API management and security to caching, developing a federation of queries on gateways, and much more.

Schema first or code first?

There are multiple ways to develop GraphQL solutions. The two most common approaches are **schema first**, in which developers write GraphQL schema first and later build client-side queries and data resolvers on the back end, and **code first** (also known as **resolvers first**), in which developers write the resolvers first and then generate the GraphQL schema for them.

Both approaches come with advantages and disadvantages based on your specific use case.

GraphQL recommendations and guidance

When working on the GraphQL section of the reference architecture, we discussed a number of principles and values that influenced the documented recommendations and guidance. Here, we'll offer a brief overview.

Schema first development

In order to support collaboration across different languages, microservices, and tools, we recommend using the GraphQL schema as a form of API definition rather than generating a schema from the code. Code-first solutions typically are limited to a single language and can create compatibility issues between the front end and other useful GraphQL tools.

Separate concerns

When your back and front-end codebase is minimal, you can use tools to generate code, analyze your schemas, and so on. Those tools typically do not run in production but provide a number of features missing in the reference architecture. All elements should work outside your application and can be replaced if needed.

Use the GraphQL reference implementation

Using the GraphQL reference implementation facilitates supportability, and it is vendor agnostic. GraphQL is a Linux Foundation project with a number of reference libraries maintained under its umbrella. Choosing these libraries over single vendor and product-focused open source libraries reduces the risk of providing support and maximizes the stability of our solutions over extended periods of time.

Minimalism

Developers often look for libraries that offer an improved API and increase productivity. In our experience, picking a high-level tool that focuses only on the essential elements needed to build a successful GraphQL API leads to the best outcome. As a result, we've chosen to include a very short list of packages and recommendations that are useful for developers.

Exclude opinionated solutions

The GraphQL section of the Node.js reference architecture does not include CRUD engines or tools that affect developer flexibility and introduce proprietary APIs.

Conclusion

Based on our discussion of these principles and values, along with our prior experience, we developed the recommendations and guidance captured in the reference architecture. We hope this chapter has given you some insight into the background and considerations the team covered in building that section. For more information, check out the GraphQL section of the Node.js reference architecture.

In the next chapter, we will discuss building good containers.

Chapter 4: Building good containers

Michael Dawson

Containers are often the unit of deployment in modern applications. An application is built into one or more container images using Docker or Podman, and then those images are deployed into production.

A container package code written in Node.js along with its dependencies can be easily deployed as a unit. The Open Container Initiative (OCI) defines the standard for what makes up a container.

This chapter dives into the discussions that went into creating the building good containers section of the Node.js reference architecture. That section focuses on how the container is built, versus how to structure an application for deployment in a container. Other sections in the reference architecture, like health checks and logging, cover how to structure an application for cloud-native deployments.

What makes a good production container?

Before we dive into the recommendations for building good containers, what do we mean by a "good" container in the first place? What this means to the Red Hat and IBM team members is that the container:

- Applies best practices for security.

- Is a reasonable size.
- Avoids common pitfalls with running a process in a container.
- Can take advantage of the resources provided to it.
- Includes what's needed to debug production issues when they occur.

While the relative priority between these can differ across teams, these were generally important based on our experience.

What base images to start with?

In most cases, teams build their containers based on a pre-existing image that includes at least the operating system (OS) and commonly also includes the runtime—in our case, Node.js.

In order to build good containers, it is important to start on solid footing by choosing a base container that is well maintained, is scanned and updated when vulnerabilities are reported, keeps up with new versions of the runtime, and (if required by your organization) has commercial support.

The reference architecture includes two sections that talk about containers: container images and commercially-supported containers. Most of the teams within Red Hat and IBM are already using or moving toward using the Node.js Red Hat Universal Base Images (UBI) for Node.js deployments.

Apply security best practices

The first thing we talked about with respect to building good containers is making sure we applied security best practices. The two recommendations that came from these discussions were:

- Build containers so that your application runs as non-root.
- Avoid reserved (privileged) ports (1–1023) inside the container.

The reason for building containers so that your application runs as non-root is well-documented, and we found it was a common practice across the team members. For a good article that dives into the details, see Processes In Containers Should Not Run As Root.

Why should you avoid using reserved (privileged) ports (1-1023)? Docker or Kubernetes will just map the port to something different anyway, right? The problem is that applications not

running as root normally cannot bind to ports 1-1023, and while it might be possible to allow this when the container is started, you generally want to avoid it. In addition, the Node.js runtime has some limitations that mean if you add the privileges needed to run on those ports when starting the container, you can no longer do things like set additional certificates in the environment. Since the ports will be mapped anyway, there is no good reason to use a reserved (privileged) port. Avoiding them can save you trouble in the future.

A real-world example: A complicated migration

Using reserved (privileged) ports inside a container led to a complicated migration process for one of our teams when they later wanted to move to a new base container that was designed to run applications as non-root.

The team had many microservices all using the same set of internal ports, and they wanted to be able to update and deploy individual microservices slowly, without having to modify the configurations outside of the container. Using different ports internally would have meant they would have to maintain the knowledge of which microservices used which ports internally, and that would make the configuration more complex and harder to maintain. The problem was that with the new base image, the microservices could no longer bind to the internal privileged port they had been using before.

The team thought, "Okay, so let's just use iptables or some other way to redirect so that even when the application binds to a port above 1023, Kubernetes still sees the service as exposed on the original privileged port." Unfortunately, that's not something that developers are expected to do in containers, and base containers don't include the components for port forwarding!

Next, they said, "Okay, let's give the containers the privileges required so that a non-root user can connect to the privileged port." Unfortunately, due to the issue in Node.js, that led to not being able to set additional certificates that they needed. In the end, the team found a way to migrate, but it was a lot more complicated than if they had not been using privileged ports.

Keep containers to a reasonable size

A common question is, "Why does container size matter?" The expectation is that with good layering and caching, the total size of a container won't end up being an issue. While that can often be true, environments like Kubernetes make it easy for containers to spin up and down and do so on different machines. Each time this happens on a new machine, you end up

having to pull down all of the components. The same happens for new deployments if you updated all of the layers starting at the OS (maybe to address CVEs).

The net is that while we've not seen complaints or had problems in our deployments with respect to the size on disk, the compressed size that might need to be transferred to a machine has led our teams to strive to minimize container size.

A common practice we discussed was multi-stage builds, where you build in a larger base container and then copy the application artifacts to a smaller deployment image. The document Use multi-stage builds provides a good overview of how to do that.

Support efficient iterative development

The discussions on keeping container sizes reasonable also resulted in a few additional recommendations from our experience that I was unaware of before. (The process of putting together the reference architecture has been a great learning experience all around.)

The first was to use the `.dockerignore` file. Once we thought about it, it made a lot of sense, as we'd run into one of the issues it addresses a number of times. If you test locally and do an `npm install`, you end up with the `node_modules` directory locally. When you run your Docker file, it will take longer, as it copies that directory over even though it won't necessarily be used in the build step (and if it is, that could mess things up). Assuming you are using a multi-stage build, it won't affect your final image size, but it does affect the speed of development as you iterate.

The second recommendation was to use a dependency image. For many applications, the build time is dominated by the time it takes to build the dependencies. If you break out your pipeline so that you build a dependency image and then layer your application into that image, the process of updating and testing the application can be much faster. This is because, for most of the iterations, you will not have updated the dependencies and can skip the slower rebuild of the dependency layer.

Build containers that can take advantage of the resources provided

The nice thing about using containers is that it decouples the application, microservice, etc., from the physical resources on which it will be deployed. It also means that the resources available to the container might change. Kubernetes, Docker, and Podman all provide ways to change the available resources when a container is started. If you don't plan or think about

this in advance, you can end up with a container that overuses or underuses the resources available to it, resulting in poorer performance than expected.

In our discussions, we found that teams had developed patterns to start Node.js applications within containers such that they could leverage the amount of memory made available when the container was deployed. The reference architecture shares this pattern as good practice so that your application leverages the available amount of resources. Since Node.js is "approximately" single-threaded, we had not found the need to pass through available CPU resources to the same extent.

Be ready to debug production issues when they occur

When things go wrong in production, you often need additional tools to help investigate what is going on. While we did not have a common set of tools to recommend from across our teams at this point, there was consensus that it is best practice to include key tools that you might need for problem investigation. This is one reason why we've been working in the Node.js project to pull some diagnostic tools into core (such as `node-report`, the ability to generate heap dumps, and the sampling heap profiler).

Avoid common pitfalls when running a process in a container

Running a Node.js process in a container is different from running on a full operating system. This results in a couple of common pitfalls related to signals, child processes, and zombies, in no particular order. Our teams ran into a number of these challenges, which resulted in the recommendations to use a process manager and avoid the use of npm start. There is not much to add here (the reference architecture provides helpful resources for further reading), other than to say these are real-world issues that one or more of our teams have run into.

Conclusion

Building good containers can result in both faster development cycles and better deployments with fewer problems. In this chapter, we've shared some of the discussion and background that resulted in the recommendations in the building good containers section of the Node.js reference architecture.

In the next chapter, we'll dive into web frameworks.

Chapter 5: Choosing web frameworks

Bethany Griggs

One of the key choices you make when building an enterprise Node.js application is the web framework that will serve as its foundation. As part of our Node.js reference architecture effort, we pulled together many internal Red Hat and IBM teams to discuss the web frameworks they've had success with. From our meetings, we learned that most of the developers we spoke to had been using Express.js. This is unsurprising since it has long been considered the default for Node.js. As a result, that is what is in the current recommendations.

Having said that, there have been concerns about the level of maintenance Express.js has had over the last number of years. On that front, there is a reinvigorated Technical Committee, and more people have started getting involved, but it’s too early to tell if that will address all of the concerns.

At the same time, there are other frameworks that are gaining on Express and might be a good fit for your deployments. The team has had success with some of them (for example, Fastify); however, there is still no clear successor to Express.

Key metrics

In looking for a successor, a few of the key metrics we looked at were:

- Weekly downloads

- npm dependents (that is, how many packages on npm depend on this module)
- GitHub dependents
- GitHub stars
- Issues
- Last release
- Creation date

You can read this article for a point-in-time review of those key metrics and leading web frameworks from back in 2021: Introduction to the Node.js reference architecture: Choosing web frameworks.

It's been a while since 2021, but based on these kinds of metrics, the team has still not identified a clear successor.

Key attributes

In making the decision on what framework to use, the team discussed a number of key attributes:

- Is it widely used? That would mean that there's a lot of shared knowledge about it both externally and within your organization.
- Can new users find a significant amount of resources to help them get started?
- Does it have a relatively shallow dependency tree?
- Is it stable— does it avoid breaking changes but still addresses security vulnerabilities as necessary?
- Is it compatible across Node.js versions?

Conclusion

Choosing a web framework based on past experience and success is a bit harder than for other components. It's too early to tell if the concerns over Express maintenance will be resolved by the recent initiatives. In that context, while the Node.js reference architecture shares our past success with Express, it's important to consider the maintenance concerns and the other growing frameworks when making your decision.

Chapter 6: Code coverage

Luke Holmquist

In this chapter, we'll look at why testing in JavaScript applications is important. You'll also learn how to measure code coverage, how to maximize your investment in testing, and what the Node.js reference architecture recommends to ensure adequate code coverage.

What is code coverage?

Code coverage is a software testing metric that determines how much code in a project has been successfully validated under a test procedure, which in turn helps in analyzing how thoroughly software has been verified.

To measure the lines of code that are actually exercised by test runs, the code coverage metric takes various criteria into consideration. The following are a few important coverage criteria.

- **Function coverage:** The functions in the source code that are called and executed at least once.
- **Statement coverage:** The number of statements that have been successfully validated in the source code.
- **Path coverage:** The flows containing a sequence of controls and conditions that have worked well at least once.

- **Branch or decision coverage:** The decision control structures (loops, for example) that have executed properly.
- **Condition coverage:** The Boolean expressions that are validated and that execute both TRUE and FALSE during the test runs.

What modules should you use for code coverage?

Assuming you've read the previous chapters, you know that we recommend modules that our teams are familiar with and use regularly. This chapter is no different. The teams defining the reference architecture have decided on two modules that we recommend for code coverage:

- nyc, probably the most popular tool for code coverage. One of the main reasons this module is the most popular is that it works well with most JavaScript testing frameworks. `nyc` is the successor command-line interface (CLI) for `istanbul`.
- Jest, which generates coverage when you run the tool with the `--coverage` option.

  Note: While `nyc` and Jest were the modules that the team had a good level of experience with, as ESM adoption grows, so do the challenges of using these modules. Newer options like C8 are something you can look at if you are using ESM.

This chapter's examples use `nyc`.

Testing example

Download the example from the Nodeshift example repository. The example is made up of two simple functions located in the `index.js` file, and a test in the `test` directory that uses the Mocha test runner.

The first function adds two numbers:

```
function addTwoNumbers(x, y) {
return x + y;
}
```

You can easily cover this function with this simple test:

```
describe('testing for coverage', () => {
  it ('should add 2 numbers correctly', () => {
    assert.equal(addTwoNumbers(1,1), 2);
  });
});
```

You can generate code coverage easily, too: Just call the `nyc` module in conjunction with our test to generate the coverage report.

The `package.json` might look something like the following to run our tests while generating coverage reports:

```
"scripts": {
    "test": "mocha",
    "coverage": "nyc npm run test"
  }
```

Executing `npm run coverage` is all that is needed. Because we wrote only one test, our statement coverage will be only about 18%. To increase code coverage, we have to test the other functions and statements:

```
  testing for coverage
    ✔ should add 2 numbers correctly

  1 passing (3ms)
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |   18.18 |        0 |      50 |   18.18 |
 index.js |   18.18 |        0 |      50 |   18.18 | 9-23
----------|---------|----------|---------|---------|-------------------
```

Once the other tests are written and run, code coverage should be at 100%:

```
testing for coverage
    ✔ should add 2 numbers correctly
    ✔ should test that phish is the best band
    ✔ should test the beatles are a good band too
    ✔ should test that nickelback is not that good
    ✔ should test that all other bands are pretty good

  5 passing (6ms)
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 index.js |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
```

What to cover

After much discussion, the reference architecture team's guidance on what to cover is broken down into two sections:

- **Coverage for modules:** It is important to cover all public APIs. Basically, any exposed API that a user can interact with should be covered.
- **Coverage for applications:** This coverage should be driven by the key user stories and key paths that the application can take. Unlike a module, here it is much more important to test and cover paths that are critical for your application than to worry about covering all paths.

Acceptable coverage thresholds

We realized in our discussions that not all projects are created equal. Some projects might be starting from scratch, whereas others contain legacy code bases.

For new projects without legacy code, a good percentage threshold is about 70%. This is because, with new projects, it is relatively easy to add tests while creating the application or module.

It might be a little harder to add coverage to a project that is older and doesn't have any coverage yet; adding tests to old code with technical debt can be a challenge, especially for someone new coming into the project. In this case, a good percentage threshold is about 30%.

It is also our experience that when adding coverage to an older code base, focusing on the key user stories gives you the best return on your investment. Focusing on the path and branch/decision coverage can also maximize the investment required to get to 30%.

Guidance for open source projects

For free and open source projects, it might be helpful to post the results of the coverage to an external service. There are multiple services available that have free open source plans as codecov.io and Coveralls. Creating issues that suggest ways to increase code coverage could be a way to attract contributors to the project.

It is also common to report the coverage increase or decrease percentage during a pull request or merge request CI run. In our experience, it is best to use this information in a code review rather than by blocking a merge.

Conclusion

In this chapter, we looked at the reasons why it is important to maximize the usefulness of our tests. We also saw what tools the Node.js reference architecture team recommends for measuring code coverage.

In the next chapter, we discuss our experience with using both plain JavaScript and TypeScript.

Chapter 7: TypeScript

Dominic Harries

One of the key choices you make when building an enterprise Node.js application is whether to use plain JavaScript or a dialect that supports type validation. While participating in the Node.js reference architecture effort, we've pulled together many internal Red Hat and IBM teams to discuss our experience with using both plain JavaScript and TypeScript. Our projects seem to be split between the two, and it's often "love it or hate it" when using types with JavaScript.

This chapter covers why you might want to use TypeScript and how to get started, along with an introduction to the recommendations in the Node.js reference architecture. As with all our Node.js reference architecture recommendations, we focus on defining a set of good and reliable default choices. Some teams will deviate from the recommendations based on their assessment of what best fits their use case.

Why use TypeScript?

JavaScript has come a long way from its humble beginnings as a lightweight scripting language inside the browser. Technologies such as Node.js have propelled it to become one of the leading languages for back-end development.

But as codebases grow in size, it can be increasingly difficult to track down errors and keep track of the data flowing through an application. That's true in any language, but it's a particular problem in weakly typed languages like JavaScript.

TypeScript is designed to address this problem. By adding type annotations to variables, TypeScript can help to document the data a program uses, catch errors, and give developers confidence that they can change code in one place without breaking other parts of their codebase.

Many code editors now have excellent TypeScript support. This support enables code completion, immediate feedback on type errors, powerful automatic refactoring, and other useful features. For instance, Visual Studio Code is a widely used editor that comes with extensive support for TypeScript. The TypeScript wiki contains a list of other editors with TypeScript support.

Most popular third-party JavaScript libraries now ship with TypeScript type definitions or make them available via the Definitely Typed repository.

These capabilities have caused TypeScript to explode in popularity.

Get started with TypeScript

TypeScript has been designed to be easy to adopt, even for existing JavaScript projects. You can incrementally enable TypeScript a single file at a time while leaving the rest of your project in JavaScript.

To demonstrate this flexibility, we'll port a very simple Node.js application to TypeScript. The application consists of a single JavaScript file named `fill.js` in the project's `src` directory. The code fills an array with a value:

```
function fillArray(len, val) {
  const arr = [];
  for (let i = 0; i < len; i++) {
    arr.push(val);
  }
  return arr;
}

module.exports = { fillArray };
```

Step 1 is to install a TypeScript compiler. Because Node.js does not natively understand TypeScript files, they must be compiled to JavaScript before they can be executed. The compilation from TypeScript to JavaScript is called transpiling. There are multiple transpilers available (see the reference architecture for details), but we'll use the standard TypeScript compiler `tsc`. Install it as follows:

```
npm install --save-dev typescript
```

If you're using any built-in Node.js modules, you also need the types for these:

```
npm install --save-dev @types/node
```

The compilation process is configured using a `tsconfig.json` file. This configuration controls all the parameters for TypeScript compilation. The Node.js community maintains a recommended configuration that you can install as follows:

```
npm install --save-dev @tsconfig/node16
```

If you are using a Node.js version older than 16, you can check the list of bases for recommended configurations compatible with older versions.

Add Node.js options to your `tsconfig.json` file as follows:

```
{
  "extends": "@tsconfig/node16/tsconfig.json",
  "compilerOptions": {
    "allowJs": true,
    "strict": false,
    "outDir": "./build"
  },
  "include": ["./src/**/*"]
}
```

This configuration specifies that all files under the `src` directory should be compiled and put in the `build` directory. It also allows your source files to stay written in JavaScript (these will be copied across to the build directory without modification) and disables strict mode (more details on strict mode later on). There are many more options that you can set—see our recommendations in the reference architecture.

To run the compile, execute:

```
npx tsc
```

In this simple example, because we haven't defined any data types, the compiler created an identical `fill.js` file in the `build` directory.

Adding some TypeScript

Node.js supports two module systems:

- **CommonJS:** The traditional format, which uses the `require` keyword to import code and module.exports to export it.
- **ES modules:** A newer format using the `import` keyword to import code and the `export` keyword to export it. This format is supported by both Node.js and web browsers.

TypeScript supports only the ES module format, so in addition to renaming your example file to `src/fill.ts`, you need to update its export:

```
export function fillArray(len, val) {
  const arr = [];
  for (let i = 0; i < len; i++) {
    arr.push(val);
  }
  return arr;
}
```

This code now compiles successfully, even though you haven't added any types. This is because strict mode is set to false in the `tsconfig.json` file. If you set the mode to true, you will see an error like the following when you compile:

```
src/fill.ts:1:27 - error TS7006: Parameter 'len' implicitly has an 'any' type.
src/fill.ts:1:32 - error TS7006: Parameter 'val' implicitly has an 'any' type.
```

You can add some annotations to the argument list in the first line to fix these errors:

```
export function fillArray(len: number, val: any) {
  const arr = [];
  for (let i = 0; i < len; i++) {
    arr.push(val);
  }
  return arr;
}
```

The changes make the compilation succeed. Even better, if you accidentally forget which way round the parameters go and call the method like this:

```
console.log(fillArray("-", 5));
```

TypeScript gives another helpful error:

```
error TS2345: Argument of type 'string' is not assignable to parameter of type
'number'.
```

We recommend enabling strict mode for new projects, but when migrating existing projects it may be easier to leave the mode disabled.

Many editors can be configured to immediately show TypeScript errors rather than waiting until you run the compiler. Editors may also offer other advanced features such as code completion and automatic refactoring.

Node.js reference architecture recommendations

Teams need to make a number of key choices when using TypeScript. These include:

- Should transpilers be used? If so, which ones?
- What should be shipped: the original files, or the transpiled versions?
- What TypeScript options and configuration should be used?
- How should types for npm packages be published?

The Node.js reference architecture contains further recommendations, including how to use TypeScript with tools such as `nodemon` and best practices for deployment based on the experience our team has gained through deployments within Red Hat, IBM, and our customers.

Those recommendations are well defined in the Node.js reference architecture, so instead of repeating them here, we encourage you to head over to the TypeScript section of the reference architecture itself.

Since the team put together the TypeScript recommendations, some interesting topics have emerged in the TypeScript space. These include:

- The user experience of using Node.js with TypeScript.
- Challenges due to interactions between ESM and TypeScript.

While we've not heard these surface in our team discussions, they are important enough to include a short discussion.

The user experience of using Node.js and TypeScript has been discussed at several Node.js collaborator summits. As you know, Node.js does not support TypeScript natively. From the collaborator discussions the best approach was agreed to be providing hooks that could be used to provided better integration versus bundling TypeScript support into Node.js

Good progress has been made on implementing "loaders" in Node.js. This article provides a good overview of how to use them to improve the developer experience when using Typescript with Node.js: Custom ESM loaders: Who, what, when, where, why, how.

Conclusion

While the choice to use TypeScript is often a "love it or hate-it" decision, we hope this chapter has helped you understand some best practices that have been successful for the team's deployments.

In the next chapter, we'll focus on key security elements that JavaScript developers should address.

Chapter 8: Securing Node.js applications

Luke Holmquist

Making your Node.js applications secure is an essential part of the development of Node.js modules and applications. Security practices apply to both the code itself and your software development process. This chapter focuses on some of the key security elements that JavaScript developers should address.

This chapter covers 8 key elements of building security into your software development process to make your Node.js applications and modules robust:

- Choosing dependencies.
- Managing access and content of public and private data stores such as npm and GitHub.
- Writing defensive code.
- Limiting required execution privileges.
- Support for logging and monitoring.
- Externalizing secrets.
- Maintaining a secure and up-to-date foundation for deployed applications.
- Maintaining individual modules.

Although this is not necessarily an exhaustive list, these are commonly the focus of the Red Hat and IBM teams.

Choosing third-party dependencies

Most Node.js applications and modules have third-party dependencies, many of which contain security vulnerabilities. Although open source teams usually fix the vulnerabilities soon after discovery, there are still gaps in time before an application developer learns about the vulnerability and puts the fixed library into production. Attackers might exploit the compromised program during those times. So, it's important to choose dependencies carefully and regularly evaluate if they remain the right choices for you.

A couple of helpful tips in this area are:

- Determine that a dependency is necessary before integrating it into your application. Is using the modules instead of your code saving development and maintenance time?
- Avoid code one-liners.
- If you have a choice of dependencies, use one that has only a few or no dependencies of its own.
- Choose dependencies that already have a high level of usage based on statistics, such as GitHub stars and npm. These tend to be maintained well.

Find more in-depth guidance on managing dependencies in the reference architecture's choosing and vetting dependencies section.

Managing access and content of public and private data stores

Modern development flows often use public and private data stores, including npm and GitHub. We recommend the following management practices:

- Enable two-factor authentication (2FA) to ensure the integrity of the committed code and published assets. GitHub, for instance, now requires a developer who logs in to verify their identity through a code sent to their device.
- Use files such as `.npmignore` and `.gitignore` to avoid accidentally publishing secrets. These are hidden files consulted by programs (npm and Git, respectively). If you list a file with your secrets in one of these hidden files, npm and Git will never check

it into the source repository. Of course, you must have a separate process to manage the secrets. There are many services available to help you.

A `.npmrc` file is often needed for npm installations, particularly if you have private modules. Avoid leaking information in the `.npmrc` file when building containers by using one of these options:

- Use two-stage builds, where you build one image with all the tools for the application and a second to create a stripped-down image. In addition to saving memory and disk space, the two-stage build allows you to omit the `.npmrc` file from the final image that goes into production.
- Avoid adding the secrets to any image in the build process. Instead, you can securely mount secrets into containers during the build process, as explained in the article How to sneak secrets into your containers. In particular, Buildah has built-in functions to make it easier to mount files with secrets.
- The least preferred method: Delete the `.npmrc` file from the final image and compress images to flatten layers.

Writing defensive code

Secure coding often calls for special training and cannot be summarized in simple precepts. Nevertheless, you can eliminate many common vulnerabilities by following the recommendations in this section. There is a more extensive list in the Secure Development Process section of the reference architecture.

Avoid global state

Using global variables makes it easy to leak information between requests accidentally. With global variables, data from one web visitor might be in memory when a second visitor sends a request. Potential impacts include corrupting the request or revealing private information to another visitor.

Each request should encapsulate its data. If you need global data, such as statistics about the traffic you are handling, store it in an external database. This solution is preferable to global variables because the data in the database is persistent.

Set the NODE_ENV environment variable to production

Some packages consult the `NODE_ENV` environment variable to decide whether they need to lock things down or share less information. Therefore, setting the variable to production is the safest setting and should be used all the time. The application developer, not the package, should determine what information to display.

Validate user input

Unvalidated input can result in attacks such as command injection, SQL injection, and denial of service, disrupting your service and corrupting data. Always validate user input before implementing it within your application code. Make sure you validate input on the server even if you validate on the client side (browser or mobile application) because an attacker could send requests directly to the APIs without using the client.

Include good exception handling

Basic practices for handling exceptions include:

- Check at a high level for missed exceptions and handle them gracefully. Make sure to have a default handler for Express and other web frameworks to avoid displaying errors with the stack trace to the visitor.
- Listen to errors when using EventEmitters.
- Check for errors passed into asynchronous calls.

Avoid complex regular expressions

Regular expressions help with text parsing tasks, such as ensuring that a visitor submitted their email address or phone number in an acceptable format or checking input for suspicious characters that could signal an attack. Unfortunately, if a regular expression is complex, it can take a long time to run. In fact, some regexes run essentially forever on certain kinds of text.

Even worse, although your regular expression might operate reasonably under most input, a malicious attacker could provide content that triggers an endless run. The article Regular expression Denial of Service - ReDoS explains this type of vulnerability.

The takeaway is to be careful about the complexity of any regular expression you use. When checking text input, avoid regular expressions or use only simple ones that check for issues such as invalid characters.

Limit the attack surface

Some helpful ways to limit the available attack surface are:

- Expose only the APIs needed to support the intended operations. For example, when using Express, remove any unnecessary routes.
- Group all external endpoints under a prefix (i.e., `/api`). This makes it easier to expose only APIs intended to be external in the ingress configuration.
- Don't rewrite paths to the root (`/`).
- Use authentication to limit access. When possible, integrate an organizational identity and access control provider instead of implementing your own.

Limiting required execution privileges

Design your applications to run with the minimum privileges required. Ensure that your applications can run as a non-root user, especially when deployed within containers. The user and group under which the application runs should have access only to a minimal set of files and resources. For more container recommendations, refer to Chapter 4: Building good containers on page 15.

Support for logging and monitoring

Logging sensitive or suspicious actions will make it easier for monitoring tools to collect and analyze the data. See the logging section of the reference architecture for recommended monitoring packages.

Externalizing secrets

Secrets (i.e., passwords) should be defined externally and made available to the application at runtime through secure means. Make sure you don't commit secrets in code repositories or build them into container images.

The article GitOps secret management provides a good overview of the techniques and components used to manage externalized secrets. The article also refers to additional resources on the topic.

More specific to Node.js deployments, consider using the dotenv package, which is popular among our team, or the newly added `--env-file` Node.js option. We also contribute to kube-service-bindings to support the Service Binding Specification for Kubernetes.

One of the leading tools for managing externalized secrets is node-vault. Teams involved in deployments with the IBM cloud find the IBM Cloud Secrets Manager Node.js SDK helpful.

Maintaining a secure and up-to-date foundation for deployed applications

A Node.js application is on top of several components. You must keep this foundation secure and up to date throughout your application's lifetime, even if no code changes within your application.

The key elements include secure and up-to-date:

- Base container images
- Node.js runtime
- Dependencies

Based on the team's experience, here are some recommended tips:

- Take advantage of container images that come with Node.js already bundled in. The maintainers usually release an update after fixing a CVE reported against the Node.js runtime or any other components within the container. This is one of the reasons the team members often use the ubi/nodejs container images.
- If you build Node.js binaries into a base image, subscribe to and read the nodejs-sec mailing list. This low-volume mailing list provides advance notice of security releases and will give you the earliest warning to update your Node.js version.
- If you use common dependencies across many projects, create a dependency image from which each project reads. While this centralization is suitable for build times, as outlined in the dependency image section of the reference architecture, it also helps reduce the total work required for dependency updates when shared across numerous projects.

For a more exhaustive list of tips, check out the Secure Development Process section of the reference architecture.

Maintaining individual modules

When you maintain modules in GitHub, enable Snyk integration and review the pull requests it creates.

It is also important to test and ensure the module runs and passes tests on the latest Long Term Support (LTS) version of Node.js. Automated testing reduces risk when Node.js security releases require updates.

Conclusion

From choosing and vetting third party libraries to externalizing your secrets and limiting your applications' attack surface, this chapter showed the importance of writing and maintaining secure code.

The next chapter covers key questions that Node.js developers need to understand about accessibility.

Chapter 9: Accessibility

Michael Dawson

Making applications accessible to disabled users is both good business and often required by law. We must develop the Node.js components that are part of these applications in a way that facilitates accessibility. This chapter covers the key questions that Node.js developers need to understand about accessibility:

- Do Node.js developers need to worry about accessibility?
- Are there standards and guidelines for accessibility?
- What do I need to do as a Node.js developer?

Why Node.js developers need to provide accessibility

When the Node.js reference architecture team started discussing recommendations for accessibility, one of the accessibility leaders within Red Hat brought us up to speed. We quickly learned that the focus on accessibility is most often on the front end during the development of many applications, giving minimal attention to JavaScript running under Node.js on the back end.

For software developed with JavaScript, the UI typically falls into one of the following categories:

- Browser-based UIs delivered through JavaScript, HTML, and CSS.
- Desktop-type applications built on something like Electron.

- Command-line scripts and applications.

The first two categories are similar in using JavaScript, HTML, and CSS to create the UI, which runs in the browser or an equivalent utility. We call these categories the front end.

The front end can access business logic and data through REST APIs, WebSockets, or other API interfaces. In these contexts, the focus of accessibility is on front-end development. But Node.js developers still need to consider accessibility because the APIs they provide will affect the accessibility of the front end. (More on this topic later.)

On the other hand, Node.js developers usually build the UI for a command-line script or application, so they are responsible for making the UI accessible.

Do Node.js developers need to worry about accessibility? Absolutely! If you develop command-line scripts or applications, you are responsible for ensuring their accessibility. You might not own the responsibility for applications where you provide the back end, but you still need to support the effort.

Are there standards and guidelines for accessibility?

There is good news and bad news on this front. Web applications with a front end can follow well-defined standards and guidelines, including the Web Content Accessibility Guidelines (WCAG). On the other hand, accessibility in command-line tools has not historically been considered a problem, and we are unaware of any standards to help.

Our team's experience is that it is helpful for organizations to define guidelines and/or to design systems that apply to their product or organization to help ensure that accessibility is part of the interface from the start. Defining your guidelines allows the team to develop its interfaces more efficiently and achieve better consistency in their accessibility. You can find specific examples of guidelines and design systems listed in the accessibility section of the Node.js reference architecture.

What should Node.js developers do?

If you are a Node.js developer working in a team building an application with a front end, you should code your APIs in conformance with the requirements of the front-end team. In addition, you should:

- Avoid hard-coding resources returned to the front end and ensure that your APIs can provide all required forms of a resource. Resources required by the front end might

need an alternative format to support accessibility (i.e., alternative text for audio playback).

- Ensure that error messages are understandable when read aloud. Error messages returned to users can be read by screen readers.

If you are building command-line scripts or applications that integrate visual elements such as color, review the corresponding sections of the web accessibility guidelines that apply.

Although there are no standards or guidelines specific to command-line scripts or applications, the paper Accessibility of Command Line Interfaces provides several good recommendations, including the following:

- Ensure that an HTML version of all documentation is available.
- Provide a way to translate long outputs into another accessible format.
- Document the output structure for each command.
- Provide a way to translate tables in command-line interface (CLI) output into another accessible format.
- Ensure that all commands provide status and progress indicators.
- Ensure that all status and progress indicators used are friendly for screen readers.
- Ensure that error messages are understandable when read aloud.

In addition, some tools for accessibility testing are written in Node.js. As a result, Node.js developers might have to assist in automating accessibility testing.

Check out the accessibility section in the Node.js reference architecture for details.

Conclusion

Making applications accessible to disabled users is both good business and often required by law. We hope this chapter has helped you understand what you should be thinking about as Node.js developer and provided some pointers to get you started.

Next, read about typical development workflows that the Node.js team has encountered.

Chapter 10: Typical development workflows

Michael Dawson

Not all developers work in the same environment or use the same workflow. It's important to understand typical development workflows when building tools and processes or sharing your experience so that you can consider how developers using different workflows may be able or unable to apply your tool, processes, or techniques, given their situation.

This chapter covers the typical developer workflows that the team has encountered and that we often factor into our discussions on the architecture, including:

- Fully local development.
- Fully local development, container based.
- Local development of team's component, remote services in a common development environment.
- Fully remote development, container based.
- Zero-install development environment.

Assumptions about workflows

Before we dive into the workflows, the first thing the Node.js reference architecture team captured were some assumptions that apply to many of the workflows:

- The end target/deployment artifact is most often a container.
- For some platforms, deployment remains a server or virtual machine
- Applications are made of multiple components, some of which may be developed by other teams.

The second assumption acknowledges that while container-based deployment is common, in some platforms and environments, it may not be possible or desirable, so we can't yet forget more deployment in bare metal or virtual machines.

5 typical workflows

In the team's discussion of the workflows, we ended up with four base workflows and one meta-variation that can be applied to the base workflows.

Meta-variation zero install

The meta-variation is a "zero-install" environment where developers use a virtualized remote environment and use their local laptop only as a thin client. This can range from simply remoting into virtual machines with `ssh` to using a cloud-based IDE, which restricts the functionality available to the developer.

The key consideration when addressing developers in a zero-install variation is that they might have much less control over their environment and might need to involve other teams in order to install/leverage any of the tools or processes that you recommend.

4 base workflows

The base workflows that the team most commonly encounters include:

- Fully local development.
- Fully local development, container based.
- Local development of team's component, remote services in a common development environment.
- Fully remote development, container based.

The typical development workflows section of the reference architecture goes into specific advantages and disadvantages for each workflow, as well as a more detailed description of each variation. We won't repeat the advantages and disadvantages of each here, as you can

read in the reference architecture itself. Instead, we'll highlight some of the interesting things that came from the discussion behind that section.

Some of the interesting things that came out of the team's discussion included the following:

- An advantage of one workflow is often a disadvantage of the next.
- A common variation between flows was which elements were local on the developer's laptop versus which elements were in a remote shared environment.
- There is likely to be a fair amount of grey where organizations are partway between two of the common workflows.

Common advantages and disadvantages

The common aspects that were either an advantage or disadvantage in the discussion for each workflow included:

- Local Resource requirements.
- Total Resource requirements.
- Risk of bugs due to a mismatch between development and deployment environments.
- Management/complexity for developers.
- Dependence on network connectivity.
- Ability to debug.
- Ease of testing out changes in other components.
- Ease of keeping development/deployment environments in sync.
- Iteration time.
- Need for centralized management/support.

One aspect is often a tradeoff against one of the others. For example, as you reduce the local resource requirements, you increase the remote resource requirements and possibly the total requirements. As you increase the use of more remote resources, you increase the need for centralized management of that resource.

Local and remote workflows

The workflows move from fully local to fully remote. What we saw in our discussions is that on the fully local end, the developer has more control and flexibility but also more work to handle the complexity of managing all of the environments needed to test with external components.

As the workflow moves towards fully remote, setup and management are simplified for the developer, and requirements for local resources (i.e., size of laptop) are reduced. However, there are tradeoffs, as using a shared environment requires more external coordination, can make debugging and experimentation more difficult, and require significant investment in managing the shared environment.

The dependence on the network as you move from local to remote also is a key consideration in terms of what developers will/will not be able to do when offline.

Conclusion

We think it's important to have defined the most common variations that the team has run into in order to help our discussion/thinking process and to consider how Node.js developers at different customers might be constrained. At the same time, we agreed in our discussions that many customers exist in an in-between state, either because that is what fits them best or because they are still on their journey to the variation that they want to get to.

In the next chapter, we offer recommendations for installing and publishing modules to the npm registry.

Chapter 11: npm development

Luke Holmquist

The Node.js ecosystem is a vast landscape, and one of the most important parts of that ecosystem is its various modules. Installing these modules is an easy process, thanks to the npm registry. It is easy to download packages, create your own module, and publish it to the npm registry. The Node.js reference architecture team has recommendations for creating and publishing your modules.

Package development

When starting the creation of a new package, it is recommended to use the `npm init` command instead of hand coding to quickly and accurately create a new `package.json`. You can even accept all the defaults by providing the `-y` flag when running the following command:

```
npm init -y
```

This command can be modified to tailor the results to your specific organization's needs. To read more about this feature, check out the official npm documentation.

While name and version are the only fields that are required to publish a package, the team recommends completing a few other fields to provide more information to the user. The following sections provide details about those fields as well as other recommendations. To see the full list, check out the npm package development section of the Node.js reference architecture.

The files field

The `files` field is a list of files that you want to be published with your package. This field is handy when using a bundler to transpile code, and you only want to include that transpiled code. The team recommends not to include tests, which should reduce the package size. But you can include docs. For a list of the files that are automatically included, check out the npm docs.

The support field

The `support` field will help package maintainers communicate with and set expectations for their users about the level of support they are willing to provide on a package. The team has worked closely with the Node.js Package Maintenance Team on their recommendations for what this support field should look like. For our opossum module, we set this field to `true` and supplied our support information in a separate package-support.json file.

For more information on the `support` field, read the Package Maintenance Team statements.

The type field

The `type` field defines the module format that Node.js will use. For ESM, use `module`. For Common JS modules, use `commonjs`. For more information on how Node.js determines the difference between ESM and CJS modules, take a look at the official docs.

Specify a license

You should specify a license for your package so that people know how they are permitted to use it and any restrictions you place on it. The team has the most experience with using both the MIT and Apache-2 licenses. But if you are publishing a package on behalf of a company, it is recommended to consult your organization about their preferred licenses.

If a package is going to be unlicensed, it is recommended to set the private field to true. You can find more information on licenses and how they relate to the `package.json`.

The main field

This is the primary entry point to your package that should be a module relative to the root of your package folder. Most of the time, the default will be `index.js` if the `main` field is not set.

The scripts property

The `scripts` property is an array containing script commands run at various times in the lifecycle of your package. The key is the lifecycle event, and the value is the command to run at that point. The team recommends creating scripts that handle calling the tests, linters, and any build steps that might need to occur. The team recommends not to use a `postInstall` script if possible, as this could be a security risk.

Advice for choosing dependencies

Another important part of package development is the dependencies. The team has created a resource for choosing and vetting dependencies. This is also discussed in Choosing third-party dependencies on page 36.

During package development, it is helpful to add `package-lock.json` to source control, so all developers working on the package can install the same versions of dependencies. It is also recommended to know what Semantic Versioning (SemVer) range you are specifying for any dependency you wish to install.

Here are a few examples:

- The caret (^) will give you all the minor and patch releases when they become available.
- The tilde (~) will include everything greater than a particular version in the same minor range.
- The SemVer calculator is a great resource to assist in making this choice.

Recommendations for publishing modules to the npm registry

To publish modules to npm, the team recommends that you turn on 2-factor authentication and use automation tokens for those modules published with some type of continuous integration/continuous delivery workflow. The following sections provide recommendations for preparing your code, transpiling sources, and module versioning.

Preparing code

It is important to keep your published packages tidy. Here are a couple ways to do this:

- Specify ignore files in a `.npmignore`.

- Specify the files you only want to publish with the `files` property in your `package.json`.

As for tests and documentation, the team has no clear recommendations. For modules our team publishes, we exclude the tests, and we publish any API docs that go beyond README to GitHub pages.

Transpiling sources

A package written in another language like TypeScript should be published with transpiled JavaScript as the primary executable.

Generally, the original sources should not be required to use your published package, but it's your preference whether to publish them or not. Similar to the guidance on tests, the modules that the team publishes that are transpiled only contain the generated code.

If your module requires build steps, it is recommended to run those before you publish and not when the user installs the package. This is where you would use the `prePublish` script, as mentioned earlier.

It is important, if you are using TypeScript or generating type definitions for your package, that you add a reference to the generated type file in your `package.json` and ensure that the file is included when publishing.

Module versions

We recommend using Semantic Versioning when possible. This will make it easier for users to determine if a new module version will potentially break their code.

There are two automation tools that can help with bumping your module to the correct version. These tools will base the version bump on commit messages that follow the conventional commit standard:

- release-please
- standard-version

The `npm` CLI also provides a version command that will increase the package version. You can find all the recommendations for publishing on the npm Publishing Guidelines GitHub page.

The npm proxy/mirror technique

The npm registry is great because it allows users to easily install third-party modules with just a few commands. But there could be some additional concerns if you are part of an organization that limits internet access or you are worried that a module you depend on could potentially disappear from the public registry.

This is where having a layer between your organization and the public npm registry can help. This is commonly referred to as an npm proxy/npm mirror.

In these situations, the team recommends using a proxy/mirror when possible. The following is a list of various scenarios in which you might consider using this technique:

- You need to limit the installation of modules to only a specific set.
- You have limited network access.
- Using a proxy/mirror can provide a centralized point for scanning security vulnerabilities.
- A mirror can reduce the dependency on the public registry.
- You need to maintain a copy of a module in case it is removed from the public registry.
- You are a good npm citizen. The public registry is a free service, and npm allows updates to 5 million requests per month, which can be used up quickly with CI builds.

The proxy/mirror techniques can be used in two ways:

- For a more global solution, set the registry using the `npm` CLI:

```
npm set registry URL
```

- Use a `.npmrc` file per project if you need more fine-grained control.

You can find more information about npm proxies and mirrors in this section of Node.js reference architecture on GitHub.

The npm registry makes installing modules easy

As previously mentioned, this is just a subset of the full list of our team's advice and recommendations for installing and publishing modules to the npm registry. For the full list of recommendations, check out the npm package development section of the Node.js reference architecture.

Conclusion

npm makes it easy to create and publish modules to its registry. In this chapter, we looked at the team's recommendations on both the creation and publishing of those modules. We also covered a few techniques to help enterprise users feel more secure.

Next, learn how to investigate 7 common problems in production.

Chapter 12: Problem determination

Michael Dawson

Leveraging the experience shared in the Node.js reference architecture can help you minimize problems in production. However, it's a fact of life that problems will still occur and you need to do problem determination. This chapter covers the Node.js reference architecture team's experience with respect to how you can investigate common problems when they do occur.

7 common problems in production

Common problems include:

- Memory leaks.
- Hangs or slow performance.
- Application failures.
- Unhandled promise rejections or exceptions.
- Resource leaks.
- Network issues.
- Natives crashes.

Preparing for production problems

Investigating problems in production most often requires tools and processes to be in place and approved in advance. It is important to define the tools you'll rely on and to get approval for those tools to either already be in place or to be installed when needed. Production environments are often tightly controlled, and doing this in advance will speed up your ability to get information when problems do occur.

The team typically used one of the following to capture a set of metrics regularly in production:

- Existing application performance management (APM) offering.
- Custom solution leveraging available platform tools (for example, those provided by a cloud provider or Red Hat OpenShift).

You can read more about the suggested approach for capturing metrics in the metrics section. When those metrics have identified there is a problem, it's time to kick off the problem determination process and try to match the symptoms to one of the common problems.

APM or custom solution

From the discussion we had within the team, one of the key factors of whether you want to use an APM or custom solution is whether you are operating a service or developing an application that will be operated by your customers.

If you are operating a service where you own all deployments, an existing application performance management (APM) solution can make sense if budget is not an issue. They offer the advantage of requiring less upfront investment and leveraging a solution designed to help investigate problems. The team has had success with using Dynatrace, Instana, and New Relic.

If you develop applications that will be operated by customers, we don't recommend adding a dependency on a specific APM for problem determination. The cost can be an issue for some customers, while others may already have standardized on a specific APM.

Whichever you choose, you'll want to be ready to:

- Instrument the application in order to capture:
    - Logs

    - Metrics
    - Traces
- Generate and extract heap snapshots.
- Generate and extract core dumps.
- Dynamically change log levels.

The implementation section of the problem determination section of the Node.js reference architecture has some good suggestions on how to do these based on our experience.

Investigating specific problems

Once you have your chosen approach in place (APM or custom) and your metrics start reporting an issue, the general flow the team discussed and agreed on for problem determination was to:

1. Match the symptoms reported by traces, metrics, logs, and health checks to one of the common problems.
2. Follow a set of steps from easiest to hardest to confirm/refute the suspected problem.
3. When confirmed, capture additional information.
4. Repeat as necessary until you've narrowed it down to the right problem.

We won't repeat the content from the common problems section of the problem determination section in the reference architecture. But each problem includes:

- **Symptoms:** How to identify the problem from the traces, metrics, logs, or health checks.
- **Approach:** The steps and tools used by the team to investigate and capture more information for that problem.
- **Guidance:** Guidance and things to look out for when applying the approach suggested.

The common problems section covers the following problems based on the team's experience:

- Memory leaks.
- Hangs or slow performance.

- Application failures.
- Unhandled promise rejections or exceptions.
- Resource leaks.
- Network issues.
- Natives crash.

The information in that section for each of those problems should help you identify what problem your application is encountering based on the symptoms and then investigate to get more information as to what might be causing it to occur.

Conclusion

We hope that this quick overview of the problem determination section of the Node.js reference architecture, along with the team discussions that led to that content, has been helpful and that the information shared in the architecture helps you in your future problem determination efforts.

Next, we'll look at testing in Node.js.

Chapter 13: Testing

Luke Holmquist

In this chapter, we'll take a look at testing in the Node.js landscape and discuss the guidelines recommended by the Node.js reference architecture team.

2 testing tools

Testing is probably one of the most important parts of any software development practice. It is also one of the most "I'll get to that later" parts. And let's be honest, we never get to it later. One of the reasons that developers have an aversion to testing is trying to figure out what tools to use.

Like most tasks in software development, the answer to which is the best package to use for testing is, it depends. It depends on the type and scope of the project. Does a one-size-fits-all solution work or should you go with a solution that is a bit more granular? These are the types of questions that the team asked itself when coming up with recommendations. There are two test frameworks that the team has had success with.

- Jest is a popular testing framework from Meta. It excels at testing React and other component-based web applications, and can be used in other contexts. It is considered an opinionated framework because it provides its own set of assertions, spies/stubs/mocks, and other features (e.g., snapshot testing and code coverage) out of the box.

- Mocha is a widely used, mature (created in 2011), and stable project. While Node.js is the main area of focus and support, you can also use Mocha in a browser context. Mocha is an unopinionated framework with a large ecosystem of plug-ins and extensions.

These frameworks have their own advantages, and in the next sections, we will take a look at why you might choose one over the other.

When Jest is the better tool

Consider using Jest when:

- Testing React or component-based applications.
- Using a compile-to-JavaScript language (like TypeScript).
- Making use of snapshots.

Snapshots provide an easy way of testing output of a function and saving the result as a snapshot artifact, which can then be used to compare against as a test. As an example:

```
test('unknown service', () => {
    expect(() => {
      bindings.getBinding('DOEST_NOT_EXIST');
    }).toThrowErrorMatchingSnapshot('unknown service');
  });
```

The first time the test runs, it will create and store a snapshot of the expected exception. On subsequent runs, if the exception does not match the snapshot, it will report a test failure. This makes generating and capturing the result from an operation testing fast and easy.

When Mocha is the better tool

Consider using Mocha when:

- You want a smaller dependency tree (91 packages versus 522).
- Jest's opinions, environment proxies, and dependency upon Babel are unfavorable for your use case.

As an example of potential problems with Jest's environment proxies, Jest replaces globals in the environment in a way that can cause failures with native add-ons. For example, this simple test fails:

```
const addon = require('bindings')('hello');

describe('test suite 1', () => {
  test('exception', () => {
    expect(addon.exception()).toThrow(TypeError);
  });
});
```

Even though the add-on is throwing the expected exception, as follows:

```
static napi_value ExceptionMethod(napi_env env, napi_callback_info info) {
   napi_throw_type_error(env, "code1", "type exception");
   return NULL;
}
```

The failure reports the exception as `TypeError: type exception`.

```
FAIL  __tests__/test.js
  ● test suite 1 › exception
    TypeError: type exception
      3 | describe('test suite 1', () => {
      4 |   test('exception', () => {
    > 5 |     expect(addon.exception()).toThrow(TypeError);
        |                   ^
      6 |   });
      7 | });
      8 |
      at Object.<anonymous> (__tests__/test.js:5:18)
```

An equivalent test runs successfully with Mocha. You can review the full source for the test on GitHub.

Recommended packages to use with Mocha

Because Mocha is unopinionated, it does not ship with batteries included. While Mocha is usable without any other third-party library, many users find the following libraries and tools helpful.

Refer to the Mocha documentation and examples repository for more information on integrating with other tools.

Assertion library

Most Mocha users will want to consume a third-party assertion library. Besides the Node.js built-in assert module, Mocha recommends choosing one of the following. Both have their own plug-in ecosystems.

- chai: The most popular general-purpose assertion library, with traditional and natural language APIs available.
- unexpected: A string-based natural language API. Mocha uses `Unexpected` in its own tests.

Stubs, spies, and mocks

Many users will want a library providing stubs, spies, and mocks to aid isolation when writing unit tests. Both have their own plug-in ecosystems.

- sinon: The most popular stub, spy, and mock library, and it is mature.
- testdouble: This is a full-featured library with the ability to mock at the module level.

Code coverage

Mocha does not automatically compute code coverage. If you need it, use the following:

- nyc: The most popular code-coverage tool, and the successor CLI for `istanbul`.

For more on code coverage, refer back to Chapter 6: Code coverage or review the code coverage section of the Node.js reference architecture.

A new feature in Node core

A recent addition to Node core could also help you determine your testing strategy. A test runner is now available in Node core 18 and 16. With the release of Node 20, it is now considered stable.

The following is a basic example of how you can use this new test runner:

```
const test = require('node:test');

test('asynchronous passing test', async (t) => {
  // This test passes because the Promise returned by the async
  // function is not rejected.
  assert.strictEqual(1, 1);
```

```
});
```

Because this is a new addition, the team doesn't have guidance on this yet, but it is on our radar to reevaluate our current guidance.

A simpler method

Before wrapping up, there is one fun and helpful thing that I would like to offer. When running your tests in a CI environment, it is somewhat trivial to specify all the different Node versions you would like to test against. This isn't as easy during development time. Many developers use a Node Version Manager like nvm to switch between different versions of Node rather than just their tests.

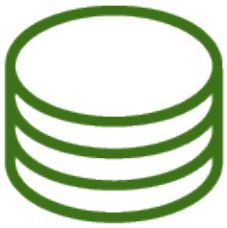
There is a slightly simpler way: using the npx command (with the `-p` flag) and the node npm module, which lets you run an npm script (or any Node application) using a different version of Node than what is installed locally. For example, to run your application's test suite with Node 18.12.1, the command might look like this:

```
npx -p node@18.12.1 -- npm run test
```

Conclusion

Testing is an integral part of application development. This chapter showed the team's recommendations on the usage of two testing tools and what situations you might choose them.

In the next chapter, we'll explore transaction handling in Node.js.



Chapter 14: Transaction handling

Michael Dawson

In many applications, completing more than a single update as an atomic unit (transaction) is needed to preserve data integrity. This chapter covers the Node.js reference architecture team's experience with integrating transactions into your application to satisfy this requirement.

Ecosystem support for transactions

Unlike the Java ecosystem, there are no well-established application servers with built-in support for transactions and no JavaScript-specific standards for transactions. Instead, Node.js applications either depend on the transaction support within databases and their related Node.js clients and/or use patterns that are needed when business logic is spread over a number of microservices. These patterns include:

- Two-phase commit
- Saga pattern

Leveraging database support for transactions

In the team's experience, if the updates that need to be completed atomically can be handled by a single Node.js component and you are using a database with support for transactions, that is the easiest case.

In some Node.js database clients, there is no specific support for transactions in the client, but that does not mean that transactions are not supported. Instead, transactions are managed by using protocol-level statements (for example, `BEGIN/COMMIT` in SQL). PostgreSQL is one of those databases. A simple example of using a transaction from the node-postgres client documentation is as follows:

```
const { Pool } = require('pg')
const pool = new Pool()

const client = await pool.connect()

try {
  await client.query('BEGIN')
  const queryText = 'INSERT INTO users(name) VALUES($1) RETURNING id'
  const res = await client.query(queryText, ['brianc'])

  const insertPhotoText = 'INSERT INTO photos(user_id, photo_url) VALUES ($1,
$2)'
  const insertPhotoValues = [res.rows[0].id, 's3.bucket.foo']
  await client.query(insertPhotoText, insertPhotoValues)
  await client.query('COMMIT')
} catch (e) {
  await client.query('ROLLBACK')
  throw e
} finally {
  client.release()
}
```

In the team's experience, transactions work best with async/await versus generators, and it is good to have a try/catch around the rollback sections in addition to those which do the commit.

The challenges of microservices

In the team's experience, the more common case is that a Node.js application incorporates a number of microservices, each of which might have a connection to the database or even may store data across different types or instances of datastores.

The challenges that come with implementing transactions with microservices are not unique to Node.js. The common approaches include:

- Two-phase commit
- Saga pattern

These are also used in Node.js applications.

The article Patterns for distributed transactions within a microservices architecture provides a nice overview of the challenges introduced when managing transactions across microservices and these two patterns.

In the team's experience, Node.js applications will most often use the Saga pattern as opposed to two-phase commit, in part because the microservices might not share the same data store.

When a two-phase commit is used, it will have to depend on external support from an underlying data store. Some databases offer support for implementing the two-phase commit technique, so read through the documentation for the database you are using if you are planning to use that technique.

Conclusion

There are no well-established application servers with built-in support for transactions and no JavaScript-specific standards for transactions. Instead, Node.js applications either depend on the transaction support within databases and their related Node.js clients and/or use patterns that are needed when business logic is spread over a number of microservices. We hope this chapter has helped you understand these common approaches so that you can implement transactions when needed.

The next chapter covers load balancing, threading, and scaling in Node.js.

Chapter 15: Load balancing, threading, and scaling

Michael Dawson

Many applications require more computation than can be handled by a single thread, CPU, process, or machine. This chapter covers the team's experience with ways to satisfy the need for larger computational resources in your Node.js application.

But isn't Node.js single threaded?

Node.js is said to be single threaded. While this is not entirely true, it reflects that most work is done on a single thread running the event loop. The asynchronous nature of JavaScript means that Node.js can handle a larger number of concurrent requests on that single thread. But if that is the case, why are we even talking about threading?

While by default a Node.js process operates in a single-threaded model, current versions of Node.js support worker threads that you can use to start additional threads of execution, each with their own event loop.

In addition, Node.js applications are often made up of multiple different microservices and multiple copies of each microservice, allowing the overall solution to leverage many concurrent threads available in a single computer or across a group of computers.

The reality is that applications based on Node.js can and do leverage multiple threads of execution over one or more computers. How to balance this work across threads, processes,

and computers and scale it in times of increased demand is an important topic for most deployments.

Keep it simple

The team's experience is that, when possible, applications should be designed so that a request to a microservice running in a container will need no more than a single thread of execution to complete in a reasonable time. If that is not possible, then worker threads are the recommended approach versus multiple processes running in a single container, as there will be lower complexity and less overhead communicating between multiple threads of execution.

Worker threads are also likely appropriate for desktop-type applications where it is known that you cannot scale beyond the resources of a single machine, and it is preferable to have the application show up as a single process instead of many individual processes.

Long-running requests

The team had a very interesting discussion around longer-running requests. Sometimes, you need to do computation that will take a while to complete, and you cannot break up that work.

The discussion centered around the following question: If we have a separate microservice that handles longer-running requests and it's okay for all requests of that type to be handled sequentially, can we just run those on the main thread loop?

Most often, the answer turns out to be no, because even in that case, you typically have other APIs like health and readiness APIs that need to respond in a reasonable amount of time when the microservice is running. If you have a request that is going to take a substantial amount of time versus completing quickly or blocking asynchronously so other work can execute on the main thread, you will need to use worker threads.

Load balancing and scaling

For requests that are completed in a timely manner, you still might need more CPU cycles than a single thread can provide in order to keep up with a larger number of requests. When implementing API requests in Node.js, they are most often designed to have no internal state, and multiple copies can be executed simultaneously. Node.js has long supported running multiple processes to allow concurrent execution of the requests through the Cluster API.

Most modern applications run in containers, and often, those containers are managed through tools like Kubernetes. In this context, the team recommends delegating load balancing and scaling to the highest layer possible instead of using the Cluster API. For example, if you deploy the application to Kubernetes, use the load balancing and scaling built into Kubernetes. In our experience, this is just as efficient or more efficient than trying to manage it at a lower level through tools like the Cluster API.

Threads versus processes

A common question is whether it is better to scale using threads or processes. Multiple threads within a single machine can typically be exploited within a single process or by starting multiple processes. Processes provide better isolation, but also lower opportunities to share resources and make communication between threads more costly. Using multiple threads within a process might be able to scale more efficiently within a single process, but it has the hard limit of only being able to scale to the resources provided by a single machine.

As described in earlier sections, the team's experience is that using worker threads when needed but otherwise leaving load balancing and scaling to management layers outside of the application itself (for example, Kubernetes) results in the right balance between the use of threads and processes across the application.

Conclusion

Node.js and the environment Node.js applications are deployed to provide a number of options for scaling through the use of multiples, threads, or a combination of the two. We hope this chapter has helped you understand which options have worked best for our team's deployments.

In the next chapter, we discuss continuous integration/continuous delivery (CI/CD) in the Node.js landscape and the guidelines recommended by the Node.js reference architecture team.

Chapter 16: CI/CD best practices

Luke Holmquist

In this chapter, we'll take a look at continuous integration/continuous delivery (CI/CD) in the Node.js landscape and discuss the guidelines recommended by the Node.js reference architecture team.

Guidance

The team's discussion on CI/CD practices was based on the common developer workflows section of the Node.js reference architecture. We found that most targets end up as a container image. Therefore, most of the guidance in this chapter is related to building and testing code that results in a container image.

During the team's discussions, we determined there were two flows that were used the most frequently together:

- Testing on code check-in.
- Using a container pipeline.

Code check-in

If you are familiar with using GitHub and GitHub Actions, this recommendation and workflow should come as no surprise. When a pull request (PR) is made against a main branch in a source code repository, this automatically kicks off initial testing using services like Travis-CI or the aforementioned GitHub Actions.

This level of testing usually starts with running unit tests and code quality checks. You can view the team's recommendations on the Node.js components to use in the code consistency, testing, and code coverage sections of the reference architecture.

Check-in testing is often configured to run on a number of Node.js versions in parallel and the team recommends that you test at least on the Long Term Support (LTS) version of Node.js that you currently deploy with along with later LTS versions in order to enable future upgrades.

It should be noted that while the check-in testing might not run on the image that flows through the container pipeline, the team tries to ensure that testing is on the same environment architecture as used by the container pipeline. In addition, the team also recommends that the same is true for local testing done by developers.

Container pipeline

While tools like Source-to-Image, Docker, or Podman can be used by the development team to test locally, they do not usually push that image to source control. Instead, the images are built as part of a container pipeline.

This workflow uses common tools such as Jenkins and Tekton to create pipelines to build and test images as they are promoted from development to production. Once updates have passed the initial check-in testing, the container pipeline kicks off. In the team's experience, the pipeline can support a number of stages or environments, including the following:

- **Development:** This environment often mirrors the main branch of each component.
- **Staging:** This stage uses tagged versions of each component that are known to work together. The team has also used multiple staging environments to mirror either a production environment or an environment that will target future production configurations.
- **Pre-production:** This is an optional environment that allows customers to sign off on images ready for production. Many teams use staging instead of separate pre-production environments, but the choice is up to your development team. That image is typically pushed to an internal registry, and later steps in the pipeline use a tagged version of that image from the registry.

- **Production:** This is the environment that hosts the customer-facing services and deployments. As the image for the Node.js component is built and promoted through the pipeline stages, it is important that all configuration related to the environment is externalized so that environment-specific values can be provided in each environment. This includes, for example, configuration for how to connect to databases and other services, the Node.js run configuration (production/development), and any other configuration information.

While a Node.js developer might not need to set up the CI/CD pipeline, a good understanding of your organization's pipeline is often valuable in order to understand what's needed to externalize configuration in the code you write, and to investigate problems that can occur during different environments in the pipeline.

Security scans

Security checks are an important part of the CI/CD workflow. Typically, the team deploys code and image scans in the code check-in tests and/or the container pipeline. The benefit of running in the code check-in tests helps developers validate that they have resolved any reported issues.

Tools like Mend.io, Snyk, and those built into GitHub are used for scanning in the check-in phase. Often, a number of different scans are required to cover all of the important aspects, which include scans of the following:

- Application dependencies for vulnerabilities.
- OS packages for vulnerabilities.
- Source code using static analysis.
- Container images for best practices (not running as root, etc.).

Looking to the future

The team periodically looks at new workflows for the CI/CD process, and something that has been announced recently is the Red Hat Trusted Software Supply Chain. It is still early in its life, but the team is looking into how this offering could fit into our guidance.

Conclusion

While not entirely Node.js-specific, this chapter showed that the Node.js reference architecture team recommends that it is important to have some sort of CI/CD pipeline that automates your applications testing and deploying.

In this book's final chapter, we'll provide a wrap-up of the Node.js reference architecture.

Wrapping up

Luke Holmquist

When we started the Node.js reference architecture journey, we wanted a way to present the team's opinion on what components our customers and internal teams should use when building Node.js applications and guidance for how to be successful in production with those components.

We also had these four points in mind during our discussions:

- Where possible, our recommendations are based on what we've used internally and in our customer engagements.
- The components specified will be our first priority for contributing to open source projects in the JavaScript ecosystem.
- Due to the preceding points, these are the components the team is best positioned when working with internal and external customers. However, we do not include formal support for these components in any of our support offerings unless specifically identified in those offerings.
- The recommended components might change over time as technologies and approaches evolve.

What we covered

Our goal was to cover a wide range of components from the team's experiences. We were able to accomplish this succinctly by breaking them up into three sections.

- Functional components
- Development
- Operations

We hope that the experience and recommendations that we've shared in these areas help you be successful in your Node.js deployments.

What's next

Now that we have "completed" version 1.0 of the Node.js reference architecture, which was mostly focused on the back end, we are now examining the front end with the web reference architecture.

We plan on following the same set of guidelines that we came up with for the Node.js back-end version. We also plan to use the same top-level headings; you can see those here.

Follow the Red Hat Node.js team

To learn more about what Red Hat is up to on the Node.js front, check out our Node.js topic page on Red Hat Developer.

About the authors

Michael Dawson is an active contributor to the Node.js project and chair of the Node.js Technical Steering Committee (TSC). He contributes to a broad range of community efforts including platform support, build infrastructure, and Node-API, as well as tools to help the community achieve quality with speed (e.g., jobs, benchmarking, and code coverage reporting). As the Node.js lead for Red Hat and IBM, he works with Red Hat's and IBM's internal teams to plan and facilitate their contributions to Node.js and v8 within the Node and Google communities.

Past experience includes building IBM's Java runtime, building and operating client-facing e-commerce applications, building PKI and symmetric-based crypto solutions as well as a number of varied consulting engagements. In his spare time, he uses Node.js to automate his home and life for fun.

Bethany Griggs is a Senior Software Engineer at Red Hat, now focusing on Backstage and Red Hat Developer Hub initiatives. With a background in software engineering and active involvement in the Node.js community since 2016, she began her career at IBM on their Node.js Runtime Team. Bethany has served on the Node.js Technical Steering Committee and the Node.js Release Working Group. Committed to open source technologies, she frequently speaks at industry conferences, sharing insights on Node.js and software development to enhance developer experiences.

Dominic Harries was previously an EMEA Dev Lead at IBM Cloud Garage.

Luke Holmquist is a senior software engineer at Red Hat. He is focused on the Node.js developer experience on OpenShift as part of the NodeShift project. While he is not a Node.js expert, he does pretend to be one at work.

James Lindeman is a software engineer at IBM.

Wojciech Trocki is a principal software engineer at Red Hat.