The Definitive Guide for Node.js in Enterprise

Insights, patterns, and best practices helping you succeed with Node.js at scale

PLATFORMATIC

Table of contents

I

01

Introduction

The basics of Node.js, who we are, and why we wrote this book

01

Introduction

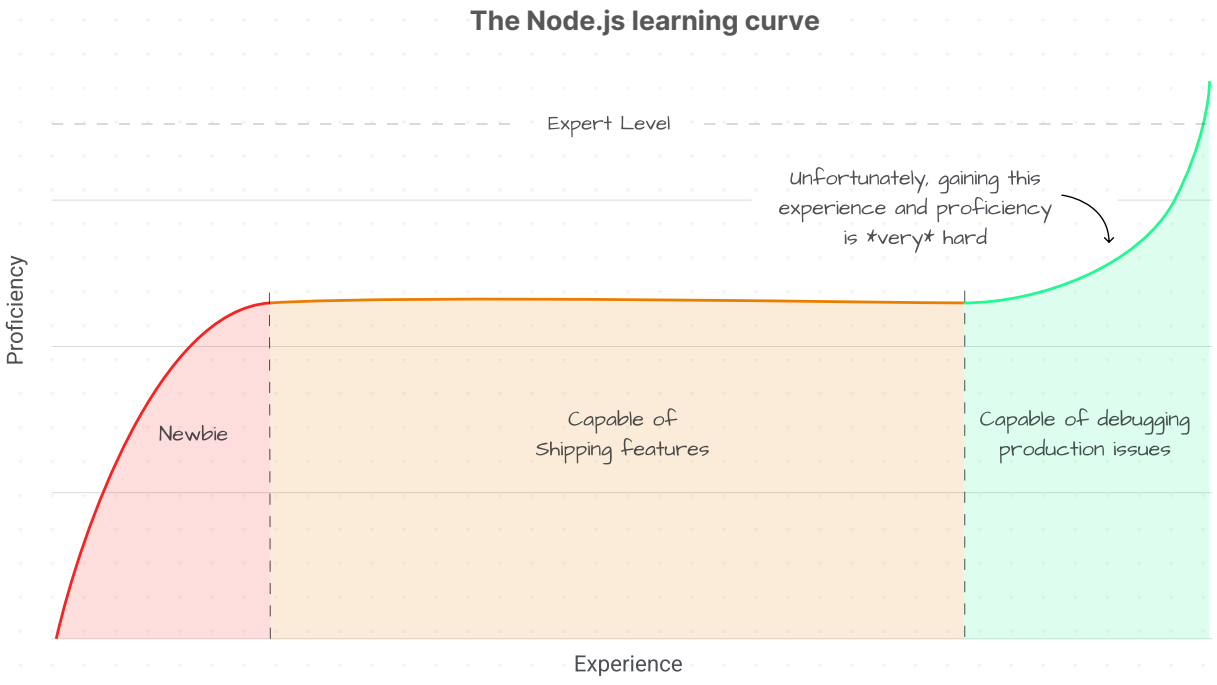
The basics of Node.js, who we are, and why we wrote this book

With years of experience helping Fortune 500 companies build and scale their most critical Node.js applications, we've seen firsthand the challenges teams face when trying to streamline their operations.

From troubleshooting distributed systems to optimizing for performance, we've tackled some of the toughest challenges Node.js has to offer. However, we know that not every team has the bandwidth, resources, or in-house expertise to do the same.

That's why we wrote this book—to share the insights, patterns, and best practices we've developed over the years, helping companies succeed with Node.js at scale.

1.1 Node.js: Easy to Learn, Hard to Master



One of the most common misconceptions about Node.js is that it's simple.
In some ways, that's true: getting a basic application up and running is remarkably easy. The lightweight nature of Node.js, its vast ecosystem, and the accessibility of JavaScript make it an appealing choice for companies looking to move fast.
But while learning the basics is straightforward, mastering Node.js is an entirely different challenge.

As companies grow and applications scale, teams often struggle with the complexities that arise. Efficient event-driven architectures, proper concurrency management, security best practices, and optimizing performance under heavy loads are just a few of the challenges that separate a working Node.js application from an enterprise-grade one.

Many teams, particularly those composed of junior developers or engineers without deep experience in Node.js, face significant hurdles in maintaining and scaling applications. A lack of best practices can lead to inefficient patterns, excessive reliance on synchronous code, and performance bottlenecks. Debugging large-scale distributed systems, understanding the event loop deeply, and handling memory management effectively require specialized knowledge that isn't always easy to acquire.

Moreover, hiring seasoned Node.js engineers is notoriously difficult. The demand for experienced developers far outpaces supply, leaving many organizations struggling to build strong backend teams. As a result, companies often resort to workarounds, patchwork solutions, and ad-hoc performance optimizations that can ultimately slow development and delay go-to-market timelines.

1.2 Our Mission at Platformatic

At Platformatic, we've taken all our expertise and poured it into both building a powerful enterprise-ready platform for Node.js and sharing our knowledge with the broader community. Our goal is to help teams navigate the complexities of Node.js with confidence in enterprise settings.

This book is a reflection of that mission. Whether you're an engineering leader looking to streamline development workflows, a senior developer seeking to fine-tune your application's performance, or a junior developer eager to level up your skills, this book will serve as a practical guide to mastering Node.js in an enterprise environment.

Let's get started.

Luca Maraschi

CO-FOUNDER & CEO, PLATFORMATIC

Luca Maraschi started coding at the age of six on a Commodore 64. By eight, he was writing code in C and decided to dedicate his career to pursuing his passion for engineering and alchemy.

Prior to co-founding Platformatic, he founded and exited three companies and held tech leadership roles at mobileLIVE and Telus Digital.

15+

years of building and operating enterprise applications

TELUS CTO.ai RCS LoyaltyOne

3

Companies built & exited

Forbes

Tech Council Member

Matteo Collina

CO-FOUNDER & CTO, PLATFORMATIC

With over 15 years of software engineering experience, Matteo is known throughout the Open Source community for his work authoring the Fastify web framework, the fast logger Pino, and his contributions to Node.js.

Matteo is a member of the Node.js Technical Steering Committee, and is an active Open Source author in the JavaScript ecosystem, with modules he maintains being downloaded over 12 billion times per year.

10+

years of building Node.js platforms for Fortune 500 companies

22 B+
Downloads per year

500+
npm modules

12 K+ Stars
25 M / month

22 K+ Stars
5.7 M / month

Node.js TSC and OpenJS Foundation Board Member

02

The Road to Node.js

JavaScript: The Foundation of Modern Web Development

02

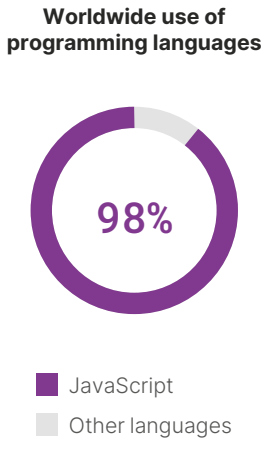
The Road to Node.js

JavaScript: The Foundation of Modern Web Development

JavaScript is the most popular programming language globally, with over 98% of websites using it for client-side programming.

Initially developed by Netscape in 1995, JavaScript was confined to web browsers but has since evolved into a versatile powerhouse, now driving everything from web applications to server-side development.

Its adaptability and ubiquity make JavaScript an ideal language for enterprise transformation.



JS

What is JavaScript?

JavaScript is one of the three core technologies used to develop websites, alongside HTML and CSS. While HTML provides structure and CSS controls styling, JavaScript adds functionality and interactivity, enabling dynamic and engaging user experiences.

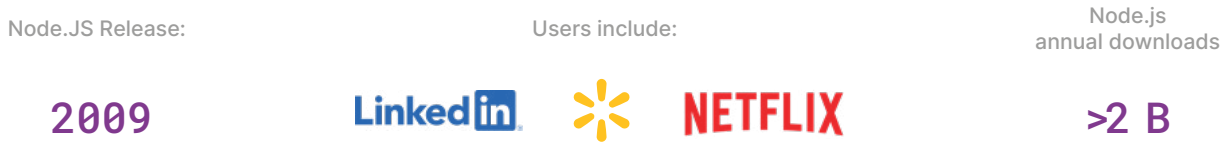
Originally a client-side scripting language, JavaScript has grown into a key player in server-side programming, mobile app development, and even desktop applications. Its lightweight syntax and event-driven nature make it both easy to learn and highly effective in a variety of contexts, contributing to its widespread adoption.

Over time, JavaScript has surpassed Java, Flash, and other languages due to its open ecosystem, strong community support, and extensive package registry.

2.1 Node.js: Bringing JavaScript to the Server

Since its release in 2009, Node.js has come a long way from being a simple niche technology.

Today, it is an effective cornerstone for modern applications including Walmart, LinkedIn and Netflix, and has over 2 billion annual downloads[1].



Node.js is a free, open source, cross-platform JavaScript runtime environment that lets developers build servers, web apps, command line tools and scripts.

Its lightweight, event-driven architecture makes it particularly well-suited for handling I/O-heavy tasks, such as web applications, Application Programming Interfaces (APIs), and microservices.

By leveraging an event-driven, non-blocking I/O model, Node.js enables applications to handle thousands of concurrent connections with minimal resource overhead.

This scalability and efficiency make Node.js an ideal choice for building high-performance, data-intensive applications that can scale to meet the demands of modern enterprise environments at a fraction of the time and cost of the previous generation of technologies such as Java or .NET, while striking a balance between performance and developer experience

This, coupled with its active and ever-growing open source community and the strong support from the OpenJS foundation– which provides a neutral home for hosting, governing, and providing financial support for essential open source JavaScript projects– has made it a pillar of contemporary web development.

[1] https://openjsf.org/

2.2 Enter npm

What truly accelerated the adoption of Node.js was npm, its built-in package manager, which provided a vast ecosystem of reusable JavaScript modules.

Released in 2010, npm allowed developers to access an ever-growing library of open-source packages, making it possible to build scalable and feature-rich applications with minimal setup.

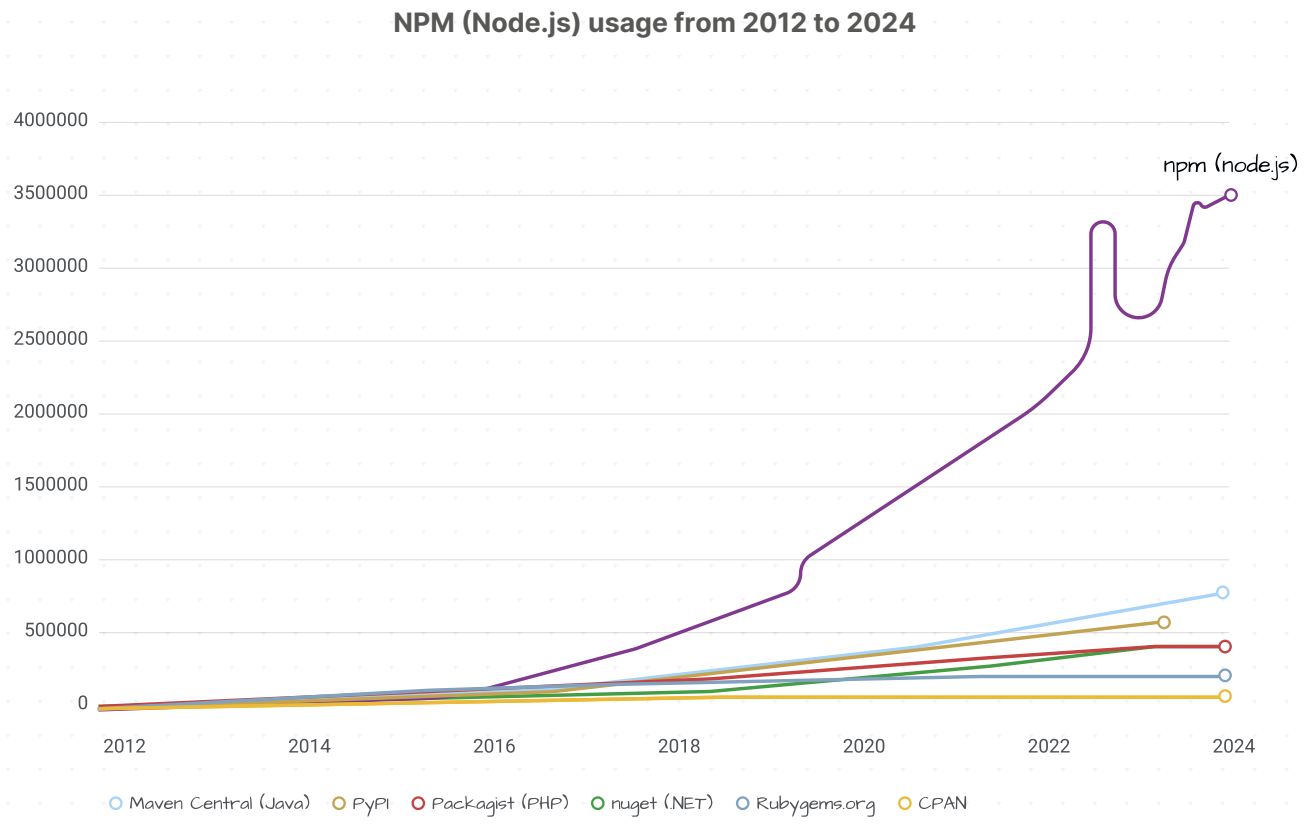
Prior to npm, developers often found themselves constantly reinventing the wheel. Common functionalities needed to be built from scratch for every project, leading to wasted time and duplicated code. npm fundamentally changed this paradigm by establishing a centralized repository for pre-written, modular code components.

These components, known as packages, encapsulate reusable functionalities that developers can easily integrate into their applications.

2010
npm Release

2012
Microsoft develops TypeScript

2015
Microsoft develops Visual Code Studio



2.3 TypeScript: Adding Static Types and Structure to JavaScript

As JavaScript continued to grow in popularity, developers began encountering challenges with scaling large applications due to the lack of strong typing. At the time, JavaScript was at a disadvantage compared to statically typed languages like Java and .NET, which had deep integration with enterprise-grade IDEs like Eclipse and Visual Studio.

JavaScript's dynamic typing did not provide enough metadata for advanced autocompletion and type inference, making it harder to scale in large projects. Recognizing this limitation, Microsoft developed TypeScript (2012) alongside Visual Studio Code (2015) to enhance JavaScript's developer experience. TypeScript brought stronger tooling, type safety, and maintainability, while Visual Studio Code provided built-in support for TypeScript, improving developer productivity through intelligent autocompletion and inline type checking.

One of TypeScript's key advantages is its ability to introduce optional static typing, allowing developers to define variable and function types at compile time. This not only improves code validation but also significantly enhances autocompletion, making development more efficient and less error-prone.

Benefits of TypeScript for Enterprise Development

Better autocompletion and developer experience
TypeScript provides richer type information, enabling more precise and intelligent autocompletion in modern IDEs like Visual Studio Code. This significantly improves coding speed and accuracy.

Improved code maintainability
Static typing makes code easier to understand for both the original developer and anyone working on the project later. Clear variable definitions enhance code readability and reduce the risk of bugs.

Enhanced developer productivity
With static type checking, errors are caught early, preventing runtime issues and allowing developers to focus on feature development.

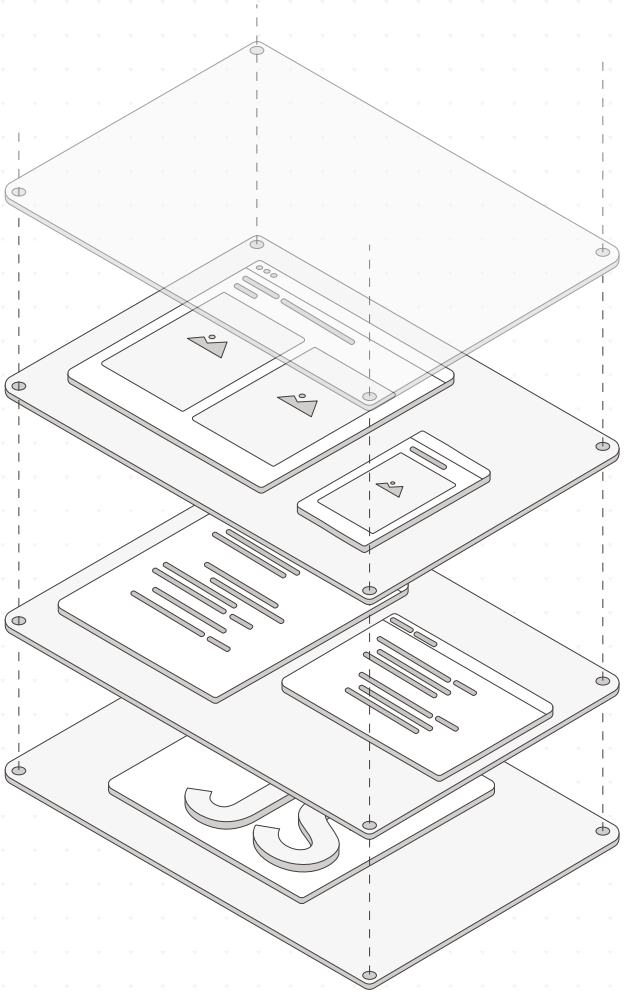
Support for large-scale applications
As projects grow in complexity, TypeScript's type system ensures better structure, code quality, and consistency across teams.

2.4 The Rise of Full-Stack JavaScript

As Node.js matured, companies embraced its event-driven architecture, making it well-suited for real-time applications, APIs, and microservices.

Meanwhile, the rise of JavaScript frameworks such as Express.js and Fastify helped streamline server-side development.

On the frontend, the introduction of React (2013) and other modern frameworks accelerated the full-stack JavaScript movement, further solidifying Node.js as the backbone of modern web development.



This full-stack shift made it possible to use JavaScript across the entire development stack, unifying development efforts and allowing developers to focus on delivering end-to-end solutions without needing to switch between multiple languages.

At the time, JavaScript code that could execute on both the client and server was coined “isomorphic JavaScript” by Charlie Robbins, Co-Founder & CEO of Nodejitsu[3].

The ecosystem surrounding full-stack JavaScript, bolstered by the npm package manager, allowed developers to access reusable modules and frameworks, accelerating development even further.

[3] https://www.oreilly.com/content/renaming-isomorphic-javascript/

Benefits of using Node.js in enterprise applications?

Unifies the development stack

By using JavaScript on both the client and server sides, organizations can streamline development processes, reduce context switching, and leverage existing skills and resources more effectively.

Additionally, Node.js promotes modularity and code reuse, enabling teams to build and maintain complex applications with greater efficiency.

Accelerates development via the small team concept

Unlike traditional development environments where developers are often siloed into specific roles, Node.js encourages cross-functional collaboration and flexibility.

Node.js's lightweight and intuitive syntax make it easy for developers to learn and use, allowing team members to quickly adapt to new tasks and technologies.

This agility enables small teams to accelerate development cycles, iterate rapidly, and respond to changing business needs with agility.

Bolsters efficiency through simple software reusability

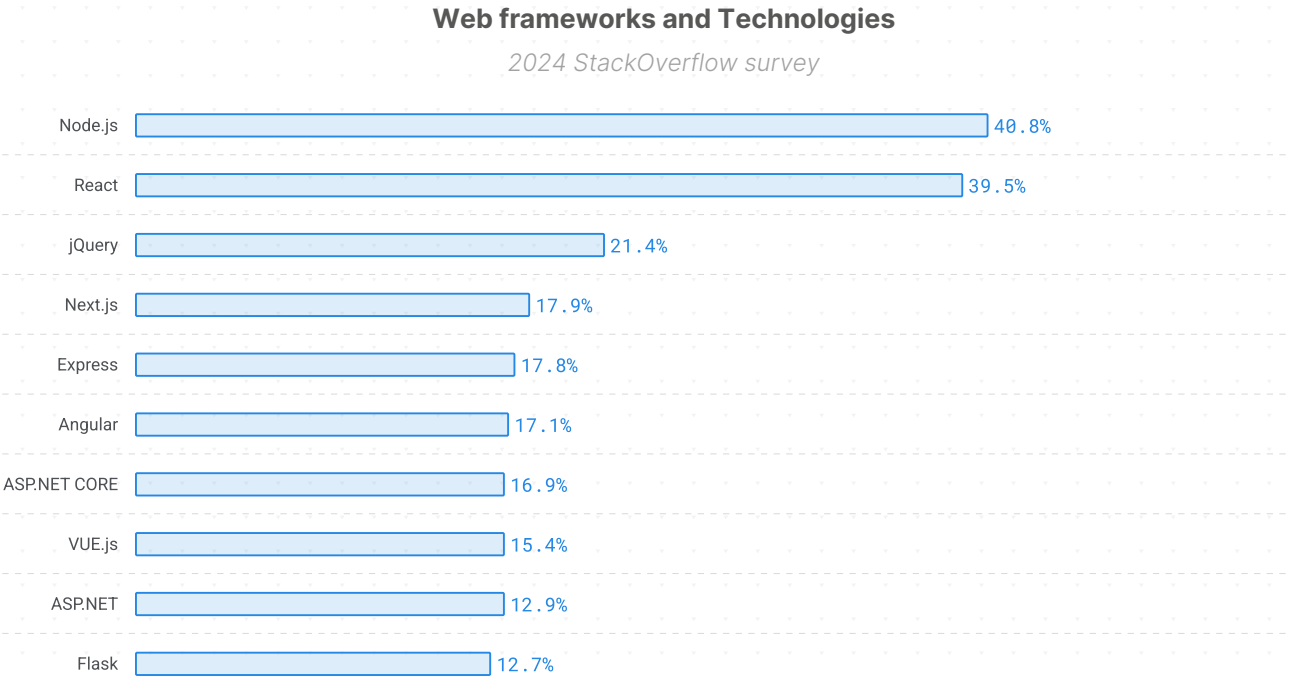
npm is a key component of the Node.js ecosystem, offering a vast repository of open source modules and packages. npm solves one of the biggest challenges in software development: code reuse.

By leveraging existing modules, developers can accelerate development, reduce duplication of effort, and maintain higher code quality.

Enterprises that adopt npm-centric development practices can code faster, iterate more efficiently, and deliver high-quality applications with greater consistency.

2.5 Node.js Usage and Growth

According to the 2024 StackOverflow survey, Node.js is the most used technology.



This widespread adoption can be attributed in large part to the powerful synergy between Node.js itself and the npm registry.

This duo has revolutionized the way developers approach software creation by tackling a major hurdle: software reusability at a massive scale.

It changed the game for developers by reducing redundancy, accelerating development, and fostering an environment of collaboration and shared knowledge. This made Node.js not only a tool for small scripts but a robust framework for building large-scale, production-ready applications.

Node.js sees a whopping 130 million downloads every month. However, it's important to understand what this number includes. A significant portion of these downloads are actually headers. These headers are temporary files downloaded during the npm i command to compile binary add-ons. Once compiled, the add-ons are stored on your system for later use.

Node.js activity over the past 10 years:

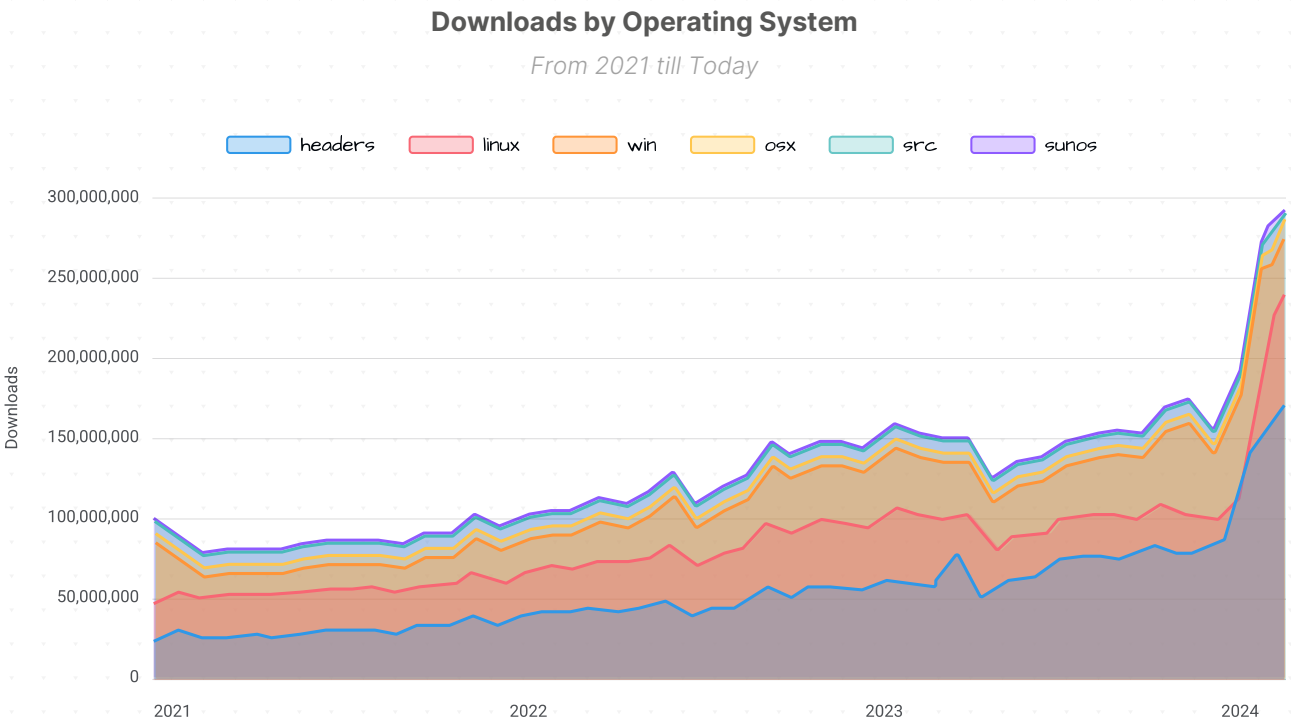
>130 M
Monthly downloads

>104,000
GitHub stars

>111,000
Commits

>27,900
Forks

>4,300
PR creators



Node.js Binaries downloads

30 M

2021 downloads

50 M

2024 downloads

>800 M

Node.js image on Docker Hub downloads

Looking at the downloads by operating system, Linux is at the top of the leaderboard.

This makes sense because Linux is often the preferred choice for continuous integration (CI) – the automated testing process software goes through during development.

While Linux dominates CI, open source projects (OSS) often perform additional tests on Windows for good measure.

This trend of high downloads translates to real usage. In 2021, there were 30 million downloads of Node.js binaries, and that number jumped to 50 million in 2024.

In 2023, the Node.js image on the Docker hub received over 800 million downloads, providing valuable insight into how much Node.js is used in production.

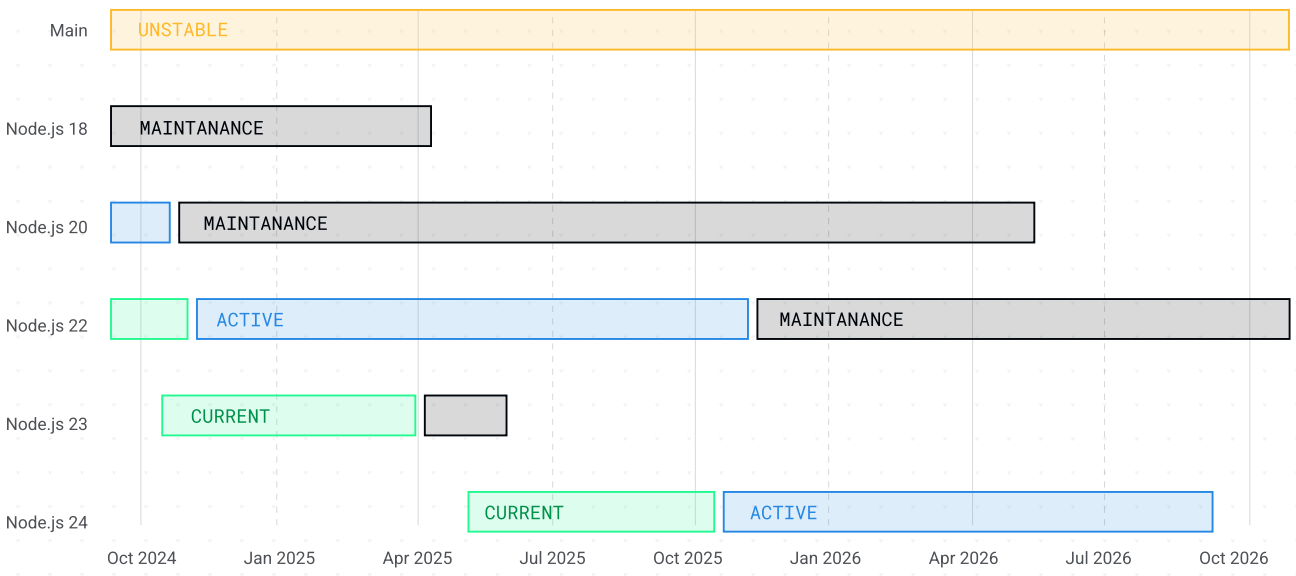
2.6 Best Practice for Node.js Maintenance

Many developers and teams are unintentionally putting their applications at risk by not updating Node.js. Here's why staying current is crucial.

Node.js offers a Long-Term Support (LTS) schedule to ensure stability and security for critical applications. However, versions eventually reach their end-of-life, meaning they no longer receive security patches.

This leaves applications built with these outdated versions vulnerable to attacks.

Long-Term Support Schedule[4]



While unfortunately we see many developers using deprecated versions of Node.js, there's good news: updating Node.js is easy.

The recommended approach is to upgrade every two LTS releases. For instance, if you're currently using Node.js 16 (which is no longer supported), you should migrate to the latest LTS active version, which is currently Node.js 22.

Don't let outdated software expose your applications to security threats.

[4] https://nodejs.org/en/about/previous-releases

2.7 Node.js' Governance

Node.js core collaborators maintain the nodejs/node GitHub repository. The GitHub team for Node.js core collaborators is @nodejs/collaborators.

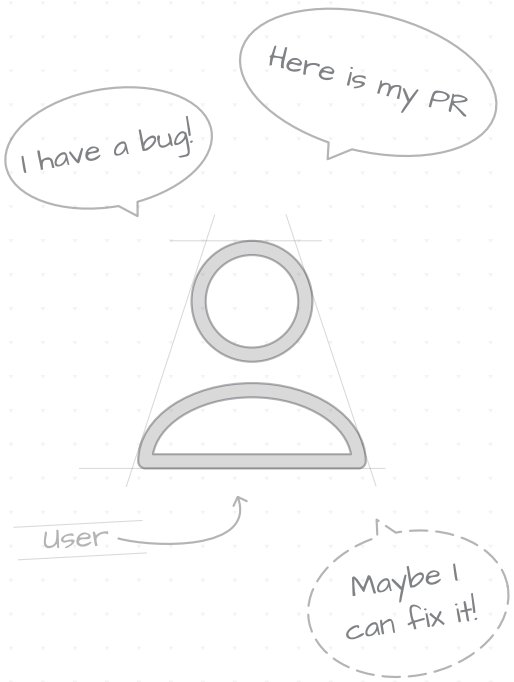
Collaborators have:

- Commit access to the nodejs/node repository
- Access to the Node.js continuous integration (CI) jobs

Both collaborators and non-collaborators may propose changes to the Node.js source code. The mechanism to propose such a change is a GitHub pull request. Collaborators review and merge (land) pull requests.

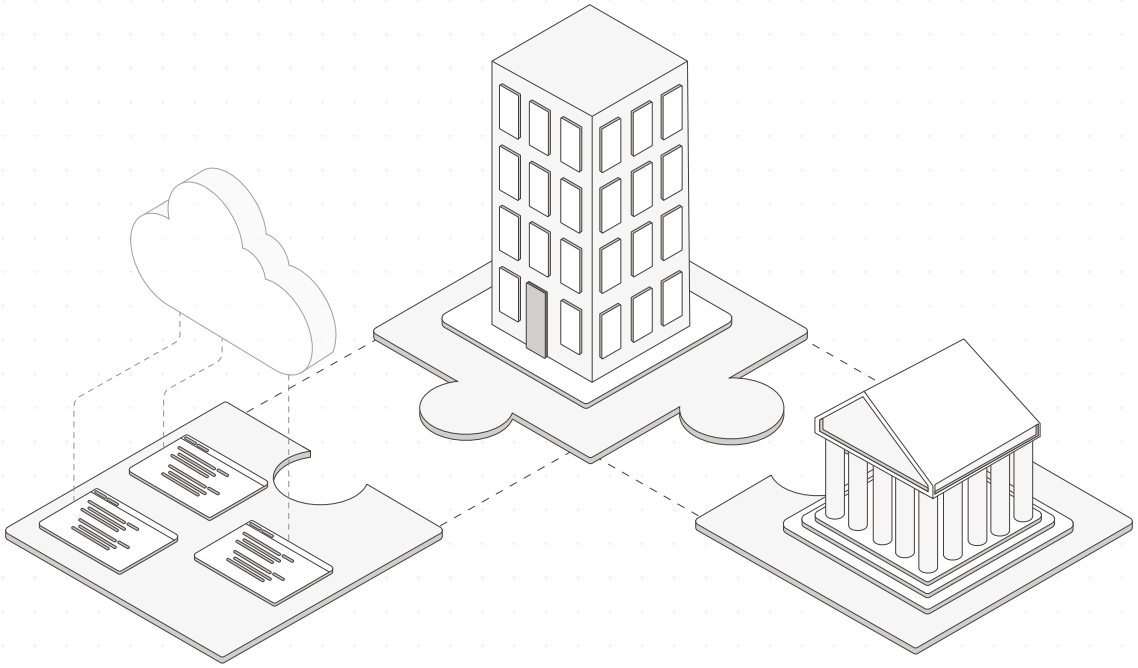
Two collaborators must approve a pull request before the pull request can land (one collaborator approval is enough if the pull request has been open for more than 7 days). Approving a pull request indicates that the collaborator accepts responsibility for the change. Approval must be from collaborators who are not authors of the change.

If a collaborator opposes a proposed change, then the change cannot land. The exception is if the TSC votes to approve the change despite the opposition.
Usually, involving the TSC is unnecessary.
Often, discussions or further changes result in collaborators removing their opposition.
Fundamentally, if you'd like to have a say in the future of Node.js, start contributing!
The project is governed by the Technical Steering Committee (TSC) which is responsible for high-level guidance of the project.



The TSC is a subset of active Collaborators who are nominated by other existing TSC members. In terms of composition, there must at all times be at least 5 members from 3 different companies.

Moreover, no one company can control more than 1/3 of the TSC.



2.8 The Open Source & Open Governance Advantage

The open source and open governance nature of Node.js provides significant advantages for enterprise teams. Unlike proprietary, vendor-locked solutions, Node.js is not just open source—it is also community-driven, meaning users have a responsibility to contribute back to the ecosystem rather than just benefit from it.

Flexibility & Customization
Open source solutions offer greater flexibility compared to traditional software with restricted control. Node.js allows enterprises to modify and extend its codebase to fit their specific needs, ensuring seamless integration with existing infrastructure and business requirements.

Innovation & Sustainability Through Open Governance
Unlike many open source projects controlled by a single company, Node.js operates under an open governance model, which means key decisions are made collectively by the community, TSC, and contributors. This structure fosters:

- **Faster innovation**
  Enterprises can actively contribute features and improvements rather than waiting on a vendor's roadmap.

- **Long-term sustainability**
  Projects benefit from diverse contributions and long-term stability because governance is not controlled by a single entity)

- **Transparent decision-making**
  Enterprises can engage directly in discussions, influence the project's direction, and ensure it aligns with their needs.

Building, Not Just Consuming: The Responsibility of Open-Source Users

The Node.js ecosystem thrives on collaboration. While enterprises benefit from the extensive community-driven enhancements, bug fixes, and libraries, they also have a responsibility to contribute back.

Simply consuming open-source software without contributing—often called "open-source vampirism"—can weaken the ecosystem over time.

Enterprises should:

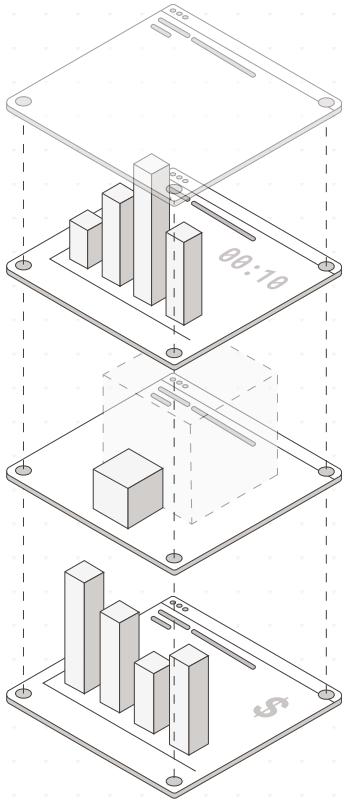
- Report and fix bugs to ensure project stability.
- Sponsor contributors or maintainers to support sustainability.
- Share improvements by contributing code back to the community.
- Engage in governance by participating in discussions and decision-making.

A Thriving Community & Global Talent Pool

The open source model attracts a diverse, global developer community, providing enterprises with access to a wide talent pool. Node.js developers often demonstrate strong problem-solving skills and collaborative mindsets, making them valuable assets for any engineering team.

By embracing both open source and open governance, enterprises not only gain technical advantages but also contribute to the long-term health of the ecosystem—a win-win for both businesses and the developer community.

2.9 Node.js Usage in Enterprise Settings



In today's business landscape, large enterprises are constantly seeking ways to improve agility, scalability, and performance while keeping an eye on costs.

Open source technologies are increasingly attractive for their cost-effectiveness, flexibility, and access to a vast collaborative developer community.

Here's a glimpse into the diverse industries leveraging Node.js to create exceptional user experiences:

Financial Services
Leading financial institutions like American Express (Amex) are adopting Node.js to build modern, data-driven applications for online banking, fraud detection, and real-time analytics. The open source nature of Node.js aligns well with the industry's growing focus on transparency and collaboration.

E-commerce
Retail giants like Walmart have utilized Node.js to create highly responsive and dynamic e-commerce platforms. Node.js allows them to handle massive user traffic during peak seasons and personalize shopping experiences in real-time.

Media & Entertainment
Streaming services like Netflix rely on Node.js for their high-performance backends, ensuring smooth video streaming and fast content delivery. Node.js efficiently manages concurrent user requests and facilitates seamless scaling as user bases grow.

Social Media
Social media platforms require real-time updates and constant user interaction. Node.js empowers these platforms to handle millions of concurrent connections and deliver a dynamic, engaging user experience.

Logistics & Transportation
Ride-hailing apps like Uber leverage Node.js to manage real-time location tracking, route optimization, and efficient rider-driver matching. Node.js facilitates smooth information exchange between various parts of the platform, ensuring a seamless user experience.

A top-tier web experience is essential across industries, from banking to e-commerce. Modern businesses rely on full-stack frameworks powered by JavaScript on the server, making Node.js a necessity for staying competitive. Below is a short checklist of key considerations you should factor in when adding Node.js to your stack.

- [ ] **Analyze organizational readiness:**
Assess the readiness of your development team to adopt Node.js. Determine whether developers have the necessary skills and expertise to work with Node.js, or if additional training or external support will be required.

- [ ] **Consider performance and security:**
Evaluate the performance and security implications of adopting Node.js. Assess potential performance bottlenecks and security vulnerabilities, and develop strategies to mitigate these risks.

- [ ] **Plan changes:**
Develop a comprehensive strategy that outlines the steps involved in adding Node.js to your stack. Consider factors such as project timelines, resource allocation, and potential impact on existing workflows.

- [ ] **Establish monitoring and support mechanisms:**
Implement monitoring and support mechanisms to ensure the stability and reliability of Node.js applications. Establish clear communication channels for reporting issues and providing support to developers.

- [ ] **Ensure you are safeguarding your code:**
The power of modularity in Node.js comes with a responsibility: ensuring the security of your application's building blocks – the module. Modern Node.js applications rely heavily on third-party modules from sources like npm. While these modules offer a wealth of functionality, they also introduce potential vulnerabilities into your application if compromised.

  To safeguard your code, it's essential to incorporate security best practices, such as using dependency scanning tools like Snyk, npm audit, and Socket to detect vulnerabilities early.

2.10 Getting started with Node.js

Before diving into Node.js development, let's set up your development environment and create your first application.

This section will guide you through installing Node.js and running a basic web server.

2.11 Installing Node.js

1. Visit the official Node.js website (https://nodejs.org)

2. Download the LTS (Long Term Support) version recommended for most users

To verify the installation, open your terminal or command prompt and run:

```
node --version
npm --version
```

Both commands should display version numbers, confirming successful installation.

2.12 Creating Your First Node.js Application

Let's create a simple web server that responds with "Hello, World!":

1. Create a new project directory:

```
mkdir my-app
cd my-app
```

2. Initialize a new Node.js project:

```
npm init -y
```

This creates a package.json file with default settings.

3. Set ES modules in your package.json:

```
{
  “type”: “module”
}
```

4. Create a new file named `server.js` and add the following code:

```
import http from ‘node:http’
import { once } from ‘node:events’

const server = http.createServer((req, res) => {
  res.writeHead(200, {
    ‘Content-Type’: ‘text/plain’
  });
  res.end(‘Hello, World!’);
});

const PORT = process.env.PORT || 3000;
await once(server.listen(PORT), ‘listening’);
console.log(`Server running at http://localhost:${PORT}/`);
```

5. Run your application:

```
node server.js
```

6. Open your web browser and visit http://localhost:3000. You should see “Hello, World!” displayed.

How can Platformatic help?

For complex projects, consider the Strangler Pattern as a risk-mitigation strategy. This approach involves gradually wrapping the legacy system with a new Node.js application, slowly shifting functionality and user traffic to the new system while the legacy system remains operational.

This minimizes disruption and allows for a more controlled migration process.

Platformatic can significantly simplify this process by facilitating service integration and streamlining communication between the legacy system and your new Node.js application.

When following the Strangler Pattern, Platformatic offers:

- Reduced Complexity: No need for manual integration code.
- Automatic Schema Refresh: We keep the unified API schema up-to-date.
- Conflict Detection: Identify and resolve API conflicts early on.
- Loose Coupling: Maintains flexibility for future modifications.
- OpenAPI & GraphQL Support: Integrates seamlessly with your existing APIs.

This allows you to focus on modernizing and delivering brilliant user experiences, while we handle the rest.

We are defining the next generation platform for building, deploying and scaling enterprise Node.js apps. Our unified Node.js platform for enterprises helps with:

- Streamlining multithreading, observability & NFR management
- Intelligent scaling & efficient resource management in Kubernetes.
- Client-side caching using HTTP standards, synchronize local copies across servers & enable invalidation across local and distributed caches
- Measuring the risk of changes in distributed environments.
- Seamless client-side caching for Next.js applications
- Extracting and interpreting the Node.js metrics that matter.

Wrapping Up

The evolution of JavaScript from a client-side scripting language to a versatile, full-stack development powerhouse has transformed the landscape of modern web development.

Node.js has played a pivotal role in this transformation by bringing JavaScript to the server-side, providing enterprises with an efficient, scalable, and flexible solution for building high-performance applications. With the support of npm, TypeScript, and a thriving open-source ecosystem, Node.js has become a critical component of both small startups and large enterprises alike.

As businesses increasingly seek agility, performance, and cost-effective solutions, the adoption of Node.js offers significant advantages. It streamlines development, reduces overhead, and enables enterprises to build robust, data-intensive applications with impressive speed.

The rise of full-stack JavaScript further reinforces the importance of Node.js as a unifying force, allowing developers to work across the entire development stack without switching languages.

Furthermore, the open source nature of Node.js, governed by a collaborative community, ensures long-term sustainability, innovation, and transparency. Its growing adoption in industries such as financial services, e-commerce, media, and logistics underscores its versatility and the benefits it brings to diverse sectors.

As organizations continue to leverage Node.js for cutting-edge solutions, tools like Platformatic are further simplifying complex enterprise processes, such as legacy system migration, and enabling businesses to focus on delivering exceptional user experiences.

In embracing Node.js, enterprises are not only modernizing their technology stacks but also participating in a thriving global ecosystem, shaping the future of web development.

03

Creating APIs with Fastify

An Intro to Fastify

03

Creating APIs with Fastify

An Intro to Fastify

Fastify is a cutting-edge open source web framework for Node.js, known for its emphasis on speed, efficiency, and an excellent developer experience.

Fastify is part of the OpenJS Foundation.

3.1 Fastify Activity

Here is a snapshot of Fastify's activity since its creation in 2016:

2016	> 32 K	> 2,300	> 3,500
Fastify Created	GitHub Stars	Forks	Pull Requests

Users Include     and many more...

Back in 2015, I was working on delivering a high-performance project. My team and I ended up optimizing this project so much that we could make Node.js saturate the network cards of our EC2 VMs.

However, because all the available frameworks added too much overhead, we had to implement all the HTTP APIs using only Node.js core, making the code difficult to maintain.

It seemed that the Node.js web framework landscape was starting to stagnate and there wasn't a solution that would meet my requirements.

I realized that in order to squeeze out the best possible Node performance, I'd have to create a new framework. I wanted this hypothetical framework to add as little overhead as possible, while also providing a good developer experience to users.

I knew that the community would be critical to what I wanted to build, and that it would be a huge effort. I quickly decided that this new web framework would adopt an open governance model, and that I would start developing it once I convinced another developer to join me in this effort.

A few months later, in June 2016, while delivering a Node.js training course at Avanscoperta in Bologna, an attendee asked me how to get started working in open source.

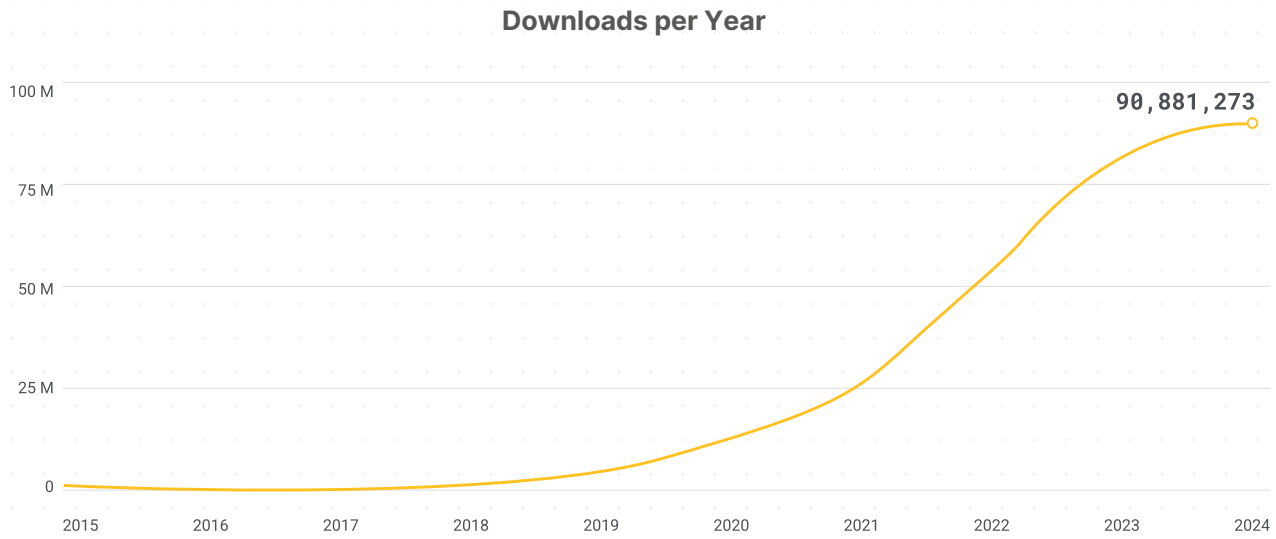
His name was Tomas Della Vedova, and by the end of the course, I asked him if he wanted to build this Node.js framework with me. By September, we landed the first commit of what would later become Fastify.

In the following years, NearForm supported me in this journey and sponsored my time for developing the framework, while Tomas was sponsored by LetzDoIt - a startup founded by Luca Maraschi.

Matteo Collina, CTO & Co-Founder, Platformatic & Co-Creator, Fastify

Usage

In 2024, Fastify was downloaded over 90 million times and doubled its downloads when compared to 2023.



Plugins Ecosystem

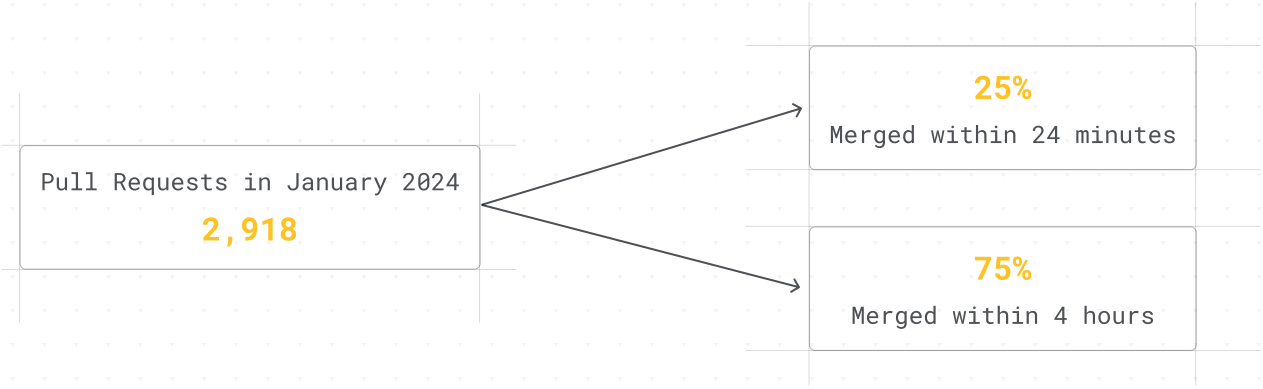
Fastify allows users to extend its functionalities with plugins. A plugin can be a set of routes, or a server decorator, among other things.

60	239
Core Plugins	Community Plugins

Maintenance

Fastify has five lead maintainers: Matteo Collina, Tomas Della Vedova, Manuel Spigolon, Kaka Ng and James Sumners, as well as an additional 8 collaborators.

The team is actively maintaining Fastify, with regular updates and a vibrant community.



3.2 Getting Started with Fastify

Fastify's journey from a high-performance experiment to one of the leading Node.js frameworks is a testament to its innovative design, speed, and flexibility.

Beyond its core capabilities, Fastify offers a robust ecosystem of tools and features that make it a strong choice for building modern APIs. In this section, we'll dive into some key aspects of working with Fastify that elevate both development and application performance.

From validating data with Ajv to simplifying test integration, handling configuration and environment variables, managing graceful shutdowns, and utilizing high-performance HTTP clients like Undici, Fastify provides a toolkit that caters to developers' needs.

Additionally, its seamless database integrations and TypeScript (via TypeBox) support ensure a streamlined workflow for building scalable and maintainable applications.

TypeBox is a TypeScript library designed to simplify the creation and validation of JSON schemas. By leveraging TypeScript's static type system, it allows developers to define data structures in a way that is both type-safe at compile-time and verifiable at runtime.

Essentially, Fastify's Typebox Integration bridges the gap between development and runtime data validation by ensuring that the data structures you define are consistent and correct throughout your application.

Let's take a closer look at these features and how they enhance the Fastify development experience.

1. First, create a new directory for your project and initialize it:

```
mkdir fastify-typebox-demo
cd fastify-typebox-demo
npm init -y
```

2. Open `package.json` and add the following configuration to enable ES modules:

```
{
  “type”: “module”,
  “scripts”: {
      “start”: “node server.ts”
  }
}
```

It’s important to note that this will work for Node.js V23+.

3. Install the required packages:

```
npm install fastify @sinclair/typebox
npm install typescript @types/node --save-dev
npm install @fastify/type-provider-typebox
```

```
import { randomUUID } from 'node:crypto';
import Fastify from 'fastify';
import { Type, type TypeBoxTypeProvider } from '@fastify/type-
provider-typebox'
export async function buildServer({ port = 3000, host = '0.0.0.0' } = {}) {
  const fastify = Fastify({
    logger: true
  }).withTypeProvider<TypeBoxTypeProvider>();
  // Define TypeBox schema for request validation
  const User = Type.Object({
    name: Type.String(),
    email: Type.String({ format: 'email' }),
    age: Type.Number({ minimum: 0 }),
    preferences: Type.Optional(Type.Object({
      newsletter: Type.Boolean(),
      theme: Type.Union([
        Type.Literal('light'), Type.Literal('dark')
      ])
    }))
  });
  // Route with TypeBox validation
  fastify.post('/users', {
    schema: {
      body: User
    }
  }, async (request, reply) => {
    const user = request.body;
    // TypeScript knows user's shape thanks to TypeBox
    return {
      message: `Created user ${user.name}`,
      userId: randomUUID()
    };
  });
  // Start server with graceful shutdown
  try {
    await fastify.listen({ port, host });
  } catch (err) {
    fastify.log.error(err);
    process.exit(1);
  }
  return fastify;
}
```

3.3 Tests with `node:test`

Introduction to Testing in Node.js

In API development, testing is a fundamental practice that ensures your application behaves as expected under various conditions. In enterprise environments, where uptime and reliability are crucial, testing becomes even more important. By writing and automating tests, you catch issues early, reduce the risk of bugs in production, and provide a reliable foundation for future changes and scaling efforts.

There are several types of testing relevant to API development:

Unit Testing
Focuses on individual functions or modules. In Fastify APIs, unit tests validate isolated route handlers or utility functions without involving other parts of the application.

Integration Testing
Ensures that different parts of the application work together correctly. For an API, this means testing routes, middleware, and interactions with databases or other services.

End-to-End (E2E) Testing
Simulates real-world scenarios from start to finish, often covering full user workflows to verify that everything works as expected in a live-like environment.

A well-tested API combines all three types of tests to achieve robust coverage. In this chapter, we'll cover how to implement unit and integration tests using Node.js's built-in node:test module to make your Fastify APIs enterprise-ready.

Overview of `node:test`

Starting with Node.js 18, the `node:test` module provides a straightforward, built-in solution for writing and running tests. It offers a simple API to write both synchronous and asynchronous tests, supports assertions, and integrates smoothly with JavaScript's `assert` module, making it a lightweight but powerful choice for API testing.

Key benefits of `node:test`

No Additional Dependencies
Since it's built into Node.js, you don't need any third-party libraries to write and run basic tests.

Easy Integration
Works well with popular libraries like Fastify, making it easy to validate API endpoints.

Flexible Assertions
The module supports a wide range of assertions, covering basic to advanced validation needs.

Let's start by setting up `node:test` for a Fastify project and writing a few initial tests.

3.4 Setting Up `node:test` for Fastify

To use `node:test`, create a dedicated folder (commonly named `tests` or `__tests__`) to store your test files. Each test file should follow a clear naming convention, such as `*.test.js` or `*.spec.js`, to make it easy to identify.

1. **Generate a Fastify project**
   Run the command to generate a Fastify application.

```
mkdir my-app
npm init fastify
npm i
```

2. Create a Test File

Create a `tests` folder and in it, create a new file named `routes.test.js` (or another descriptive name). Add the following basic structure to start:

```
import test from 'node:test';
import { equal } from 'node:assert';
import Fastify from 'fastify';
test('GET /greet returns expected message', async (t) => {
  const fastify = Fastify();
  t.after(async () => {
    await fastify.close();
  });
  fastify.get('/greet', async (request, reply) => {
    return {
      message: 'Hello, World!'
    };
  });
  const response = await fastify.inject({
    method: 'GET',
    url: '/greet'
  });
  equal(response.statusCode, 200, 'Expected status code 200');
  const body = JSON.parse(response.payload);
  equal(body.message, 'Hello, World!', 'Expected greeting message');
})
```

3. Run the Test

Run the test file with Node.js directly from the command line:

```
node --experimental-strip-types tests/routes.test.ts
```

You can add more tests by adding their paths and filenames.
Alternatively you can run all tests in a directory directly from the command line:

```
node --experimental-strip-types --test
```

This will automatically discover and run all files named `*.test.js` or `*.spec.js` in your project.

3.5 Writing Unit Tests for Fastify Handlers

In unit testing, we focus on testing individual route handlers. By isolating each handler, we ensure that each function behaves correctly without depending on external components or services.

This isolation is particularly useful for enterprise applications, where each handler may have complex logic and dependencies.

Let's say we have a more complex route that takes a name parameter and returns a personalized greeting. Here's how we can write a unit test for this handler:

```
import test from 'node:test';
import { equal } from 'node:assert';
import { buildServer } from './server.ts';
test('GET /greet/:name returns a personalized greeting', async (t) => {
  const fastify = await buildServer();
  t.after(async () => {
    await fastify.close();
  });
  const response = await fastify.inject({
    method: 'GET',
    url: '/greet/Alice'
  });
  equal(response.statusCode, 200, 'Expected status code 200');
  const body = await response.json();
  equal(body.message, 'Hello, Alice!', 'Expected personalized
  greeting');
})
```

This test verifies that our `/greet/:name` route correctly returns a personalized message. By using `node:test` and Fastify's `inject` method, we can test the route handler in isolation without starting an actual server.

3.6 Integration Testing with Fastify and `node:test`

While unit tests validate individual functions, integration tests ensure that multiple components work together.

Fastify's `inject` method is particularly useful here, allowing us to simulate full HTTP requests and validate responses.

For instance, if your Fastify application interacts with a database to retrieve user data, you can set up integration tests to validate the entire flow:

```
import { test } from 'node:test';
import { equal, deepEqual } from 'node:assert';
import { buildServer } from './server.ts';


test('User endpoint', async (t) => {
  const fastify = await buildServer();
  t.after(async () => {
    await fastify.close();
  });


  await t.test('GET /user/:id returns the correct user data', async (t) => {
    const response = await fastify.inject({
      method: 'GET',
      url: '/user/123'
    });
    equal(response.statusCode, 200, 'Expected status code 200');
    const body = await response.json();
    deepEqual(body, {
      id: '123',
      name: 'Alice'
    }, 'Expected user data');
  });
});
```

This example is an integration test for retrieving user data, and here we are fetching the user "Alice" and the respective "id." Integration tests are powerful tools for identifying issues that arise from interactions between different parts of your API.

Best Practices for Tests with `node:test`

Keep Tests Isolated

Ensure that each test is independent and doesn't rely on external state. This practice makes tests more reliable and reduces flakiness.

Use Descriptive Assertions

Always provide meaningful messages with assertions to clarify test failures.

Automate in CI/CD

Integrate `node:test` into your CI/CD pipeline to automatically run tests with each commit. This ensures that new code doesn't introduce regressions.

Advanced Unit Testing Techniques

As your Fastify application grows, route handlers may include more complex logic that requires additional testing techniques. Here are a few advanced techniques for more robust unit testing.

Testing Edge Cases and Error Handling

In production, your API will encounter unexpected inputs and failure conditions. It's essential to test edge cases and confirm that errors are properly handled and logged.

1. Error Handling in Fastify Routes
   Imagine a route that fails if required data is missing. Testing this error scenario helps verify that your error responses are clear and informative.

```
import { test } from 'node:test';
import { equal } from 'node:assert';
import { buildServer } from './server.ts';


test('Calculate endpoint', async (t) => {
  const fastify = await buildServer();
  t.after(async () => {
    await fastify.close();
  });

  await t.test('missing number triggers error', async (t) => {
    const response = await fastify.inject({
      method: 'GET',
      url: '/calculate/'
    });
    equal(response.statusCode, 500);
    const body = await response.json();
    equal(body.message, 'Number parameter is required');
  });
  await t.test('calculates correctly with valid number', async (t) => {
    const response = await fastify.inject({
      method: 'GET',
      url: '/calculate/5'
    });
    equal(response.statusCode, 200);
    const body = await response.json();
    equal(body.result, 10);
  });
});
```

In this test, a missing `number` parameter triggers an error, and the response is checked to confirm it's handled gracefully.

Creating E2E Tests with `node:test`

End-to-end (E2E) tests simulate real user flows, verifying that all parts of the system work together seamlessly. For Fastify, E2E tests might include calling multiple endpoints in sequence to simulate workflows. To set up E2E tests effectively, consider running a separate test database or using Docker to simulate production conditions.

1. Simulating a User Signup and Data Retrieval Flow

```
import { test } from 'node:test';
import assert from 'node:assert';
import { buildServer } from './server.ts';


test('User signup and profile retrieval flow', async (t) => {
  const server = await buildServer();
  await server.listen();
  try {
    const port = server.address().port;
    // Real HTTP request for signup
    const signupRes = await fetch(`http://localhost:${port}/signup`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        username: 'testuser', password: 'password'
      })
    });
    assert.equal(signupRes.status, 201);
    const { userId } = await signupRes.json();
    // Real HTTP request for profile
    const profileRes = await fetch(`http://localhost:${port}/
    user/${userId}`);
    assert.equal(profileRes.status, 200);
    const profile = await profileRes.json();
    assert.equal(profile.username, 'testuser');
  } finally {
    await server.close();
  }
});
```

In this E2E test, a user signs up, and then their profile is retrieved. These types of tests verify that multi-step workflows function as expected.

Performance Testing in Node.js Applications

While simple timing measurements using `performance.now()` can provide basic insights, they don't accurately represent real-world API performance under load. Enterprise applications need proper load testing tools to simulate concurrent users and measure throughput, latency, and error rates.

Here's how to properly load test your API using autocannon:

1. Basic Performance Benchmark
   First, add server as shown below:

```
import { buildServer } from './server.ts';


const server = await buildServer();
await server.listen({ port: 3000 });
```

You can test your load using autocannon:

```
import autocannon from 'autocannon';

async function runBenchmark() {
  const result = await autocannon({
    url: 'http://localhost:3000/heavy-route',
    connections: 100,
    duration: 10,
    pipelining: 1
  });

  console.log(autocannon.printResult(result));
}

runBenchmark();
```

Running this test produces detailed performance metrics:

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	2 ms	2 ms	4 ms	4 ms	2.58 ms	2.59 ms	145 ms

Stat Min	1%	2.5%	50%	97.5%	Avg	Stdev	Max
Req/Sec	27,567	27,567	33,759	36,223	32,779.2	3,070.99	27.565
Bytes/Sec	7.2 MB	7.2 MB	8.81 MB	9.54 MB	8.56 MB	801 kB	7.19 MB

This tests the response time for the `/heavy-route` endpoint, ensuring it meets performance expectations, and in this case a response time under 2.58ms.

Automating Tests in CI/CD Pipelines

Testing is most effective when automated. Integrate `node:test` into your CI/CD pipeline to run tests on every commit, making sure new changes don't introduce errors.

1. Setting Up GitHub Actions for `node:test`

```
name: Tests CI

on:
push:
branches:
- main

jobs:
test:
runs - on: ubuntu - latest

steps:
- uses: actions / checkout@v2
- uses: actions / setup - node@v2
with:
node - version: '22'
  - run: npm install
    - run: npm test
```

With this GitHub Actions workflow, tests run automatically on every push to the main branch, ensuring code quality and reliability.

3.7 Config Handling and Environment Variables

Configuration handling is important for separating application code from environment-specific details, like database credentials or API keys. A good configuration strategy ensures your application is flexible and secure, making it easy to adjust settings for different environments, such as development, testing, and production.

Why Environment Variables Matter in Enterprise Applications

Environment variables help keep sensitive information and application settings secure, preventing credentials or secrets from being hard-coded directly into your application. By loading configurations at runtime, environment variables enable seamless deployments across various stages of your application lifecycle.

Environment isolation
Each environment (development, staging, production) can have its own variables.

Enhanced security
Secrets are kept outside the codebase.

Better scalability
Adjust settings quickly without code modifications.

Setting Up Environment Variables with `.env` Files

A `.env` file is a simple way to manage environment variables locally. Each line represents a key-value pair that the application loads at runtime. Here's an example of a `.env` file:

```
PORT=3000
DATABASE_URL=postgres://username:password@localhost:5432/mydatabase
JWT_SECRET=mySuperSecretKey
API_KEY=abc123def456
```

1. Create a .env file in your project root.
2. Add it to your .gitignore file to keep sensitive information out of version control.

Using .env file for Config Management

Node.js now includes built-in environment variable loading.
You can use the `--env-file` flag:

```
console.log(process.env.PORT);
```

Run with:

```
node --env-file=.env index.js
```

With this, you can run the application with your project environment variables without hard-coding the values.

Securing Environment Variables in Production

While `.env` files are helpful in development, they should be avoided in production environments. Instead, use your platform's secret management tools. Here are some ways to protect sensitive data in production:

Environment-specific secret managers
Like AWS Secrets Manager, Azure Key Vault, or Google Cloud Secret Manager.

Environment variables via CI/CD systems
Such as GitHub Actions or GitLab CI/CD, where secrets can be securely managed and injected during deployment.

Vault-based secrets management
For even tighter security controls.

Environment-Specific Configurations

While it's common to see `NODE_ENV` used for environment-specific configurations, this approach has serious drawbacks:

1. `NODE_ENV` should only be used for what it was designed for: telling Node.js and frameworks whether we're in production mode (enabling optimizations) or development mode (enabling debugging features).

2. Using `NODE_ENV` for custom configuration:
    - Mixes application config with runtime optimization flags
    - Makes it impossible to have production-like settings in non-production environments
    - Creates confusion when you need more environments than just "development" and "production"

Instead, use a dedicated environment variable for your application environment:

```
const config = {
  database: {
    url: process.env.DATABASE_URL,
    pool: parseInt(process.env.DATABASE_POOL_SIZE, 10)
  },
  logging: {
    level: process.env.LOG_LEVEL
  },
};

export default config;
```

Run your application with explicit configuration:

```
# Production
DATABASE_URL=postgres://prod LOG_LEVEL=error node app.js

# Staging with production-like settings
DATABASE_URL=postgres://staging LOG_LEVEL=error node app.js

# Development
DATABASE_URL=postgres://dev LOG_LEVEL=debug node app.js
```

Configuring with env-schema and Typebox

For robust configuration management in Fastify applications, use env-schema with Typebox for type-safe validation:

```
import envSchema from 'env-schema';
import { Type } from '@sinclair/typebox';

const schema = Type.Object({
  PORT: Type.String({ default: '3000' }),
  DATABASE_URL: Type.String(),
  DB_USER: Type.String(),
  DB_PASSWORD: Type.String(),
  LOG_LEVEL: Type.Union([
    Type.Literal('debug'),
    Type.Literal('info'),
    Type.Literal('error')
  ], { default: 'info' })
});

const config = envSchema({
  schema,
  dotenv: true // optional, will load .env if present
});

export default config;
```

Best Practices for Config Management in Production

- **Never commit** `.env` files to version control.
- **Limit access to secrets** using role-based access controls (RBAC) in your cloud provider or CI/CD environment.
- **Log carefully** — avoid logging sensitive environment variables at all costs.
- **Use hierarchical configuration** to provide default values and allow overrides for specific environments.

Example of Hierarchical Configuration with Default Values:

```
const config = {
  port: process.env.PORT || 3000,
  db: {
    url: process.env.DATABASE_URL || 'postgres://localhost:5432/defaultdb',
    user: process.env.DB_USER || 'defaultUser',
    password: process.env.DB_PASS || 'defaultPass',
  },
};
```

3.8 Graceful Shutdowns with close-with-grace

In modern applications, especially those running in containers or cloud environments, handling shutdowns gracefully is essential.

A graceful shutdown allows an application to terminate open connections, complete any pending work, release resources, and exit cleanly. This prevents data loss, ensures reliability, and avoids unexpected behavior, especially in high-traffic enterprise systems.

The Importance of Graceful Shutdowns in Enterprise Applications

Enterprise applications often deal with high volumes of concurrent requests, database connections, and other resources that must be managed responsibly.

Without proper shutdown handling, abrupt terminations can lead to issues such as:

Data loss or corruption
If the application closes abruptly, ongoing transactions or requests may not complete.

Resource leakage
Unreleased resources like database connections or file handles can lead to memory leaks or bottlenecks.

Poor user experience
If requests are cut off mid-process, clients may experience errors or delays.

Implementing graceful shutdowns with tools like `close-with-grace` helps manage these risks and improve application reliability.

Overview of the close-with-grace Library

The close-with-grace library provides a structured way to handle application shutdown events.

It intercepts termination signals and allows you to register cleanup logic that runs before the process exits, ensuring everything is neatly closed down.

Some features of `close-with-grace`:

Handles multiple shutdown signals
like SIGINT, SIGTERM, and SIGQUIT.

Asynchronous support
for tasks such as database disconnections or closing HTTP servers.

Customizable timeout
to specify how long to wait for all cleanup tasks to complete.

Setting Up close-with-grace in a Fastify App

Adding `close-with-grace` to a Fastify application is straightforward. Here's a basic example:

1. Install `close-with-grace`:

```
npm install close-with-grace
```

2. Set up the graceful shutdown handler in your Fastify application:

```
import Fastify from 'fastify';
import closeWithGrace from 'close-with-grace';

const fastify = Fastify({
  logger: {
    level: 'info',
    transport: {
      target: 'pino-pretty'
    }
  }
});

// Example Fastify route
fastify.get('/health', async (request, reply) => {
  return { status: 'OK' };
});
```

```
// Register the close-with-grace handler
closeWithGrace(async ({
  signal
}) => {
  fastify.log.info(`Received ${signal}. Closing application...`);
  await fastify.close();
  fastify.log.info('Application closed.');
});

// Start the server
try {
  await fastify.listen({
    port: 3000
  });
} catch (err) {
  fastify.log.error(err);
  process.exit(1);
}
```

3. Test the graceful shutdown by sending termination signals (e.g., `Ctrl+C` in your terminal) and observing the application's behavior.

Handling Open Connections and Resources

To prevent resource leakage, use `close-with-grace` to handle open connections or cleanup tasks gracefully. Examples of resources to close include:

- Database connections
- File handles
- Network connections
- Message queues (e.g., RabbitMQ)

Example: Closing a Database Connection

```
import { MongoClient } from 'mongodb';

const client = new MongoClient(process.env.DATABASE_URL);
await client.connect();

// Graceful shutdown handler
closeWithGrace(async ({ signal, err }) => {
  await client.close(); // Close the database connection
  console.log('Database connection closed.');
});
```

Managing Asynchronous Tasks on Shutdown

In some cases, an application may have ongoing asynchronous tasks that need to complete before the process exits. Here's how to handle these tasks effectively:

1. Wrap asynchronous tasks in a promise that resolves upon completion.
2. Set a timeout to ensure the process exits even if a task hangs.

Example: Close with grace shutdown with Asynchronous Cleanup Tasks

```
closeWithGrace(async ({ signal }) => {
  console.log(`Received ${signal}. Cleaning up...`);

  // Example: wait for all pending promises to resolve
  await Promise.all([
    myAsyncTask1(),
    myAsyncTask2(),
  ]);

  console.log('Cleanup completed.');
});
```

Timeouts: You can set an optional timeout to enforce an exit if tasks take too long.

```
import { setTimeout as wait } from 'node:timers/promises';

closeWithGrace(async ({ signal }) => {
  console.log(`Received ${signal}. Cleaning up...`);

  await Promise.race([
    myAsyncTask1(),
    myAsyncTask2(),
    wait(5_000)
  ]);

  console.log('Cleanup completed or timed out.');
});
```

Testing Graceful Shutdowns

While testing graceful shutdowns can be challenging, we can effectively test our shutdown logic by focusing on server cleanup and in-flight request handling:

```
import { test } from 'node:test';
import { equal } from 'node:assert';
import { buildServer } from './server.ts';


// Test basic shutdown sequence
test('Server graceful shutdown', async (t) => {
  const server = await buildServer();
  let shutdownCalled = false;
  await server.listen();
  server.addHook('onClose', async () => {
    shutdownCalled = true;
  });
  await server.close();
  equal(shutdownCalled, true);
});


// Test handling of in-flight requests during shutdown
test('Server handles in-flight requests during shutdown', async (t)
=> {
  const server = await buildServer();
  await server.listen();
  // Simulate long-running request
  const longRequest = fetch(`http://localhost:${server.address().
  port}/slow-endpoint`);
  // Initiate shutdown while request is in progress
  const closePromise = server.close();
  // Verify both request completion and shutdown
  await Promise.all([longRequest, closePromise]);
});
```

```
test('Should handle graceful shutdown', async () => {
  process.emit('SIGINT');
  await someCleanupTask(); // Simulated task
  expect(someResource).toBe(null); // Check if resources are
  released
  });
```

Best Practices for Graceful Shutdowns in Production

Implement proper cleanup hooks

a. Use Fastify's `onClose` hooks for resource cleanup
b. Ensure database connections are properly closed
c. Clean up any temporary files or resources

Handle in-flight requests

a. Allow current requests to complete
b. Stop accepting new requests
c. Set appropriate timeouts for request completion

Monitor shutdown process

a. Use Fastify's logger to track shutdown stages
b. Log completion of critical cleanup tasks
c. Monitor shutdown duration

3. 9 Clients with Undici

Undici is a high-performance HTTP client built specifically for Node.js. Developed by the Node.js core team, Undici was designed to address some of the performance and memory limitations of other HTTP clients. Key advantages include:

High performance
Optimized for speed and efficiency.

Low memory overhead
Minimal memory footprint, ideal for high-concurrency applications.

Promise-based API
Simplifies asynchronous code with modern JavaScript syntax.

Node.js core support
As an officially supported client, Undici offers compatibility and integration with Node.js core features.

These benefits make Undici an ideal choice for enterprise applications requiring efficient and fast HTTP connections, especially those handling a high volume of requests.

Setting Up Undici

1. Installation
   To start, install Undici via npm:

```
npm install undici
```

2. Basic Import
   Import Undici into your project and set up a basic client:
   `import { request } from 'undici';`

3. Environment Compatibility
   Since Undici leverages modern JavaScript features, ensure your Node.js version is 14 or above for compatibility.

Making Basic HTTP Requests

With Undici, making HTTP requests is both efficient and straightforward. Here's an example of a simple GET request to fetch data from an external API.

Example: Basic GET Request

```
import { request } from 'undici';

async function fetchUserData() {
  const { statusCode, headers, body } = await request('https://api.
  example.com/user');

  console.log('Status:', statusCode);
  console.log('Headers:', headers);

  const data = await body.json();
  console.log('User Data:', data);
}

fetchUserData();
```

Handling Advanced Request Options

Undici supports various HTTP methods (GET, POST, PUT, DELETE, etc.) and request options, making it flexible for advanced use cases. Below are some common configurations:

1. POST Request with JSON Body

```
import { request } from 'undici';

async function createUser(userData) {
  const { statusCode, body } = await request('https://api.example.com
  users', {
    method: 'POST',
    headers: {
      'content-type': 'application/json',
    },
    body: JSON.stringify(userData),
  });

  const response = await body.json();
  console.log('User Created:', response);
}

createUser({ name: 'John Doe', email: 'johndoe@example.com' });
```

2. Adding Authentication Headers

```
async function fetchProtectedData() {
  const { body } = await request('https://api.example.com/protected', {
    headers: {
      authorization: `Bearer ${process.env.API_TOKEN}`,
    },
  });

  const data = await body.json();
  console.log('Protected Data:', data);
}

fetchProtectedData();
```

Managing Responses

Undici provides direct access to the response status code, headers, and body, allowing fine-grained control over responses. Use `.body.json()` or `.body.text()` based on the response type.

Example: Handling JSON Responses with Undici

```
async function fetchData() {
  const { body } = await request('https://api.example.com/data');
  const data = await body.json();
  console.log('Fetched Data:', data);
}
```

Integrating Undici with Fastify

Fastify, a powerful web framework for Node.js, can be seamlessly integrated with Undici. This allows for efficient HTTP requests within Fastify routes.

1. Basic Route with Undici

```
import Fastify from 'fastify';
import { request } from 'undici';

const fastify = Fastify();

fastify.get('/external-data', async (req, reply) => {
  const { body } = await request('https://api.example.com/data');
  const data = await body.json();
  reply.send(data);
});

fastify.listen({ port: 3000 });
```

2. Adding Error Handling in Fastify

When integrating Undici with Fastify, handle errors properly to maintain smooth request handling.

```
fastify.get('/external-data', async (req, reply) => {
  try {
    const { body } = await request('https://api.example.com/data');
    const data = await body.json();
    reply.send(data);
  } catch (error) {
    console.error('Error fetching data:', error);
    reply.status(500).send({ error: 'Failed to fetch data' });
  }
});
```

Performance Best Practices with Undici

To maximize Undici's performance:

Reuse connections
Utilize HTTP keep-alive connections for reduced overhead in high-concurrency environments.

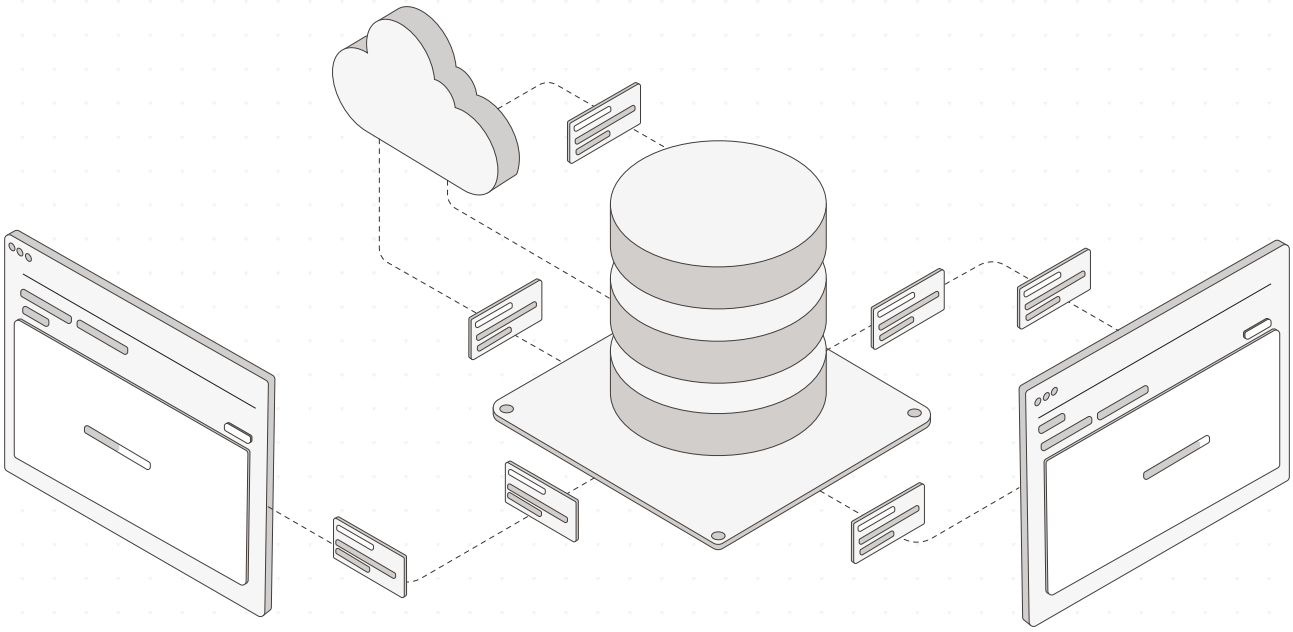
Use the Undici pool
The pool feature enables better management of multiple connections to the same host, reducing latency.

```
import { Pool } from 'undici';

const pool = new Pool('https://api.example.com');

async function fetchDataFromPool() {
  const { body } = await pool.request({ path: '/data', method: 'GET'
  });
  const data = await body.json();
  console.log('Pooled Data:', data);
}

fetchDataFromPool();
```

Error Handling and Retries with Undici

Undici does not include built-in retry mechanisms, so handling errors and retries is crucial, particularly for enterprise-grade applications.

```
import {
  request,
  getGlobalDispatcher,
  interceptors
} from 'undici';

// Configure retry behavior with the interceptor
const response = await request('https://api.example.com/data', {
  dispatcher: getGlobalDispatcher()
    .compose(interceptors.retry({
      maxRetries: 3,
      minTimeout: 1000,
      maxTimeout: 10000,
      timeoutFactor: 2,
      retryAfter: true
    }))
});

const data = await response.body.json();
```



3.10 Connecting to a database

Database integration is a cornerstone of most enterprise applications. Whether dealing with relational data in SQL databases or unstructured data in NoSQL stores, the goal is to store, retrieve, and manage data reliably and efficiently.

For Node.js applications, the asynchronous nature of database connections aligns well with Node's non-blocking architecture, allowing smooth handling of concurrent requests without major bottlenecks.

Choosing the Right Database for Enterprise Applications

Selecting the appropriate database type is crucial. Here are the main categories commonly used in enterprise applications:

Relational Databases (SQL)
Options include PostgreSQL, MySQL, and Microsoft SQL Server. These databases are ideal for structured data and complex relationships.

NoSQL Databases
MongoDB, Cassandra, and Redis are suitable for applications needing flexibility with data structure and high scalability.

In-Memory Databases
Redis and Memcached are popular choices for caching and managing real-time data.

Considerations:

Selecting the appropriate database type is crucial.

Scalability
Does the database support horizontal scaling?

Complexity
Are relational data and complex querying required?

Consistency vs. Availability
Based on the application's needs, choose between high availability or strong consistency.

Setting Up Database Dependencies

For this section, we'll demonstrate using PostgreSQL (a SQL database) and MongoDB (a NoSQL database).

1. Install PostgreSQL Driver:

```
npm install pg
```

2. Install MongoDB Driver:

```
npm install mongodb
```

3. Optional: If using an ORM or query builder like Prisma or Knex, install it as well.

Connecting to a SQL Database (PostgreSQL Example)

Let's walk through connecting a Fastify application to PostgreSQL:

1. Configuring the Database:
   Set up a PostgreSQL database and ensure you have the following credentials ready:
   - Database URL
   - Username
   - Password

2. Creating the Connection:

```
const { Client } = require('pg');
const { Pool } = 'pg'

const client = new Pool({
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT,
  max: 20,
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000,
});

async function connectToDatabase() {
  try {
    await client.connect();
    console.log('Connected to PostgreSQL');
  } catch (err) {
    console.error('Failed to connect to PostgreSQL', err);
    process.exit(1);
  }
}

connectToDatabase();
```

3. Basic Query Example:

```
async function fetchUsers() {
  const res = await client.query('SELECT * FROM users');
  console.log(res.rows);
}

fetchUsers();
```

Connecting to a NoSQL Database (MongoDB Example)

Here's how to set up a MongoDB connection:

1. Setting Up MongoDB URI:
   For local MongoDB setup:

```
MONGO_URI=mongodb://localhost:27017/yourDatabase
```

2. Creating a Connection with MongoDB:

```
const { MongoClient } = require('mongodb');

const client = new MongoClient(process.env.MONGO_URI);

async function connectToMongoDB() {
  try {
    await client.connect();
    console.log('Connected to MongoDB');
  } catch (error) {
    console.error('Failed to connect to MongoDB', error);
    process.exit(1);
  }
}

connectToMongoDB();
```

3. Fetching Data from MongoDB:

```
async function fetchUsers() {
  const db = client.db('yourDatabase');
  const users = await db.collection('users').find({}).toArray();
  console.log(users);
}

fetchUsers();
```

Database Configurations and Environment Variables

To avoid hardcoding sensitive database credentials, it's best to store configurations in environment variables. Here's an example `.env` file setup:

```
DB_USER=yourUser
DB_HOST=localhost
DB_NAME=yourDatabase
DB_PASSWORD=yourPassword
DB_PORT=5432
MONGO_URI=mongodb://localhost:27017/yourDatabase
```

Using ORM and Query Builders (Prisma and Knex)

For enterprise-grade applications, using ORMs or query builders helps with data abstraction, allowing you to manage models without raw SQL.

1. Using Prisma:

```
// schema.prisma
model User {
  id  Int  @id @default(autoincrement())
  name  String
  email  String @unique
}
```

```
npx prisma generate
```

2. Using Knex for Query Building:

```
const knex = require('knex')({
  client: 'pg',
  connection: process.env.DATABASE_URL,
});

async function getAllUsers() {
  const users = await knex.select('*').from('users');
  console.log(users);
}

getAllUsers();
```

Integrating Databases with Fastify

Fastify makes it easy to add database integrations by decorating the Fastify instance with your database client. Here's an example using PostgreSQL:

```
import Fastify from 'fastify';
import fastifyPostgres from '@fastify/postgres';

const fastify = Fastify({
  logger: true
});

// Register the official Postgres plugin
await fastify.register(fastifyPostgres, {
  connectionString: process.env.DATABASE_URL,
  // Pool configuration options
  pool: {
    max: 20,
    idleTimeoutMillis: 30000,
    connectionTimeoutMillis: 2000
  }
});

// Example route using the plugin
fastify.get('/users', async (request, reply) => {
  const { rows } = await fastify.pg.query('SELECT * FROM users');
  return rows;
});

try {
  await fastify.listen({
    port: 3000
  });
} catch (err) {
  fastify.log.error(err);
  process.exit(1);
}
```

Connection Pooling and Performance Optimization

Most SQL drivers, including PostgreSQL and MongoDB, support connection pooling, which optimizes resource use and improves response time in high-traffic applications. Here's how to set up connection pooling for PostgreSQL:

```
import Fastify from “fastify”;
import postgres from “@fastify/postgres”;

export function buildServer() {
  const fastify = Fastify({
    logger: false,
  });

  fastify.register(postgres, {
    connectionString: “postgres://localhost@localhost:5432/product_db”,
  });

  fastify.get(“/users”, async (request, reply) => {
    const client = await fastify.pg.connect();
    try {
      const { rows } = await client.query(“SELECT * FROM users”);
      return rows;
    } finally {
      client.release();
    }
  });

  fastify.get(“/user/:id”, async (request) => {
    const user = {
      id: request.params.id,
      name: “Alice”
    };
    return user;
  });

  return fastify;
}

if (import.meta.url === `file://${process.argv[1]}`) {
  try {
    const server = buildServer();
    await server.listen({
      port: 3000
    });
    console.log(“Server running on port 3000”);
  } catch (err) {
    console.error(err);
    process.exit(1);
  }
};
```

Best Practices for Secure and Resilient Connections

Use Environment Variables
Never hardcode sensitive credentials; instead, store them in `.env` files or a secrets manager.

Set Connection Timeouts
Define connection timeouts and retry mechanisms to handle transient failures.

Monitor Database Performance
Use monitoring tools like Prometheus or APM services to keep an eye on database performance.

Implement Retry Logic
Handle transient connection failures gracefully with retry mechanisms.

Limit Connection Pool Size
Avoid overwhelming your database by setting an appropriate pool size based on your application's usage.

3.11 Handle errors and provide meaningful logs

In enterprise Node.js applications, reliable error handling and logging are crucial. A robust error-handling and logging system ensures that issues are identified quickly, logs are comprehensible, and system behavior is traceable.

This approach enables efficient debugging, smoother troubleshooting, and a better developer experience, all of which are essential for high-stakes applications.

Why Error Handling Matters in Node.js APIs

Handling errors effectively:

Improves User Experience
Users receive clear, actionable error messages.

Aids Development
Developers can identify and resolve issues faster with meaningful error logs.

Increases Resilience
The system can handle unexpected failures gracefully, often without crashing.

Enhances Security
By avoiding overly detailed error messages for users, sensitive information remains secure.

Node.js applications, especially those handling asynchronous operations, require a consistent approach to catching both synchronous and asynchronous errors.

Setting Up Fastify Error Handlers

Fastify provides a built-in mechanism for handling errors with customized handlers. This example demonstrates how to set up a global error handler to respond consistently to client requests while capturing errors for logging.

1. Basic Error Handler Setup:

```
import Fastify from 'fastify';

const fastify = Fastify({
    logger: true
});

fastify.setErrorHandler((error, request, reply) => {
  request.log.error(error); // Log error for developers

  const statusCode = error.statusCode || 500;
  const response = {
    statusCode,
    error: error.message || 'Internal Server Error',
  };

  reply.status(statusCode).send(response);
});
```

Explanation:

The `setErrorHandler` method ensures that all uncaught errors are intercepted and processed, avoiding crashes and providing controlled responses.

2. Error Handling by Route:

Fastify allows specific error handling per route, which is useful for routes requiring specialized responses:

```
fastify.get('/api/resource', async (request, reply) => {
  try {
    // Business logic here
  } catch (error) {
    reply.status(500).send({ message: 'Resource could not be fetched'
    });
  }
});
```

Types of Errors in Node.js and Fastify

Errors in Node.js fall under three main categories:

Operational Errors
Known issues like database connection failures, file not found, etc.

Programming Errors
Bugs in code (e.g., undefined variables).

System Errors
Unexpected system-related issues, such as memory overflow.

Understanding these distinctions helps handle errors appropriately, using logging levels and structured error messaging.

Providing Meaningful Error Messages for Users and Developers

User-Facing Messages

- Simplify messages and provide general guidance, like “An error occurred, please try again.”
- Avoid exposing implementation details or stack traces.

Developer-Facing Messages

- Include details about the error location, the type of error, and related request information (headers, body).
- Stack traces can be captured with higher logging levels but hidden from end users.

Integrating Pino for Structured and High-Performance Logging

Fastify uses Pino as its default logger, known for its speed and structured logging capabilities. Here’s how to configure and optimize Pino for an enterprise environment:

1. Basic Configuration:

```
import Fastify from 'fastify';

const fastify = Fastify({
  logger: {
    level: 'info',
    transport: process.stdout.isTTY ? {
      target: 'pino-pretty',
      options: {
        colorize: true
      }
    } : undefined
  }
});
```

2. Logging Middleware for Route-Specific Logs:

```
fastify.addHook('onRequest', (request, reply, done) => {
  request.log.info({
    url: request.url
  }, 'Incoming request');
  done();
});

fastify.addHook('onResponse', (request, reply, done) => {
  request.log.info({
    url: request.url,
    statusCode: reply.statusCode,
    responseTime: reply.getResponseTime()
  }, 'Request completed');
  done();
});
```

3. Error Logging with Pino:

Using Pino's structured logging, errors are easily readable and consistent. By default, Fastify logs errors through Pino, but error details can be extended with custom properties:

```
fastify.setErrorHandler((error, request, reply) => {
  request.log.error({
    msg: 'Error occurred',
    errorType: error.constructor.name,
    route: request.url,
    payload: request.body,
    stack: error.stack,
  });

  reply.status(500).send({ error: 'An internal error occurred' });
});
```

Error Monitoring and Logging Aggregation Tools

For robust error handling and logging, use monitoring and aggregation tools that work well with Pino and Fastify.

These tools provide analytics, real-time alerts, and visualization:

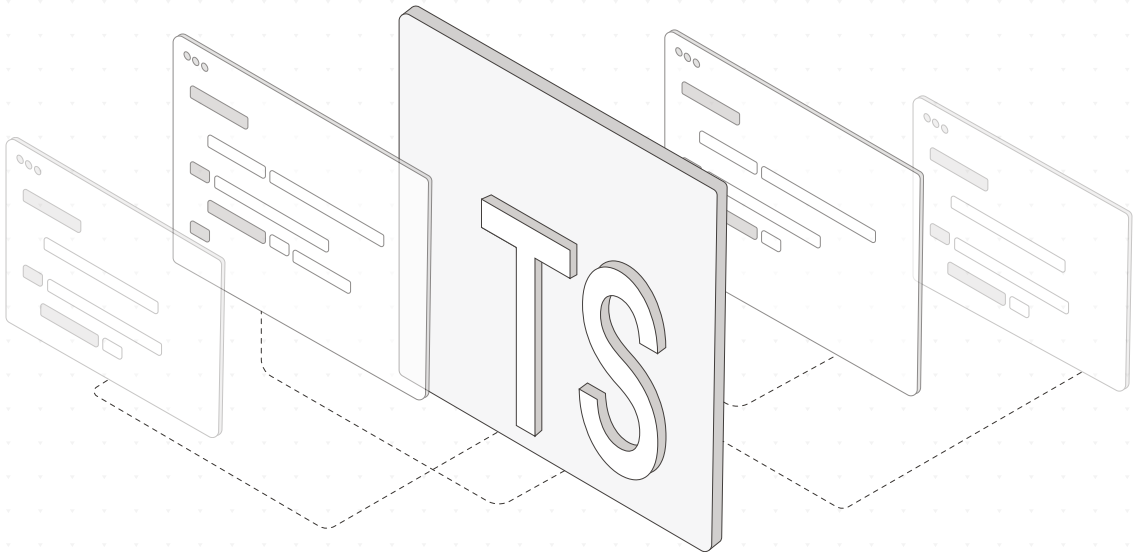
Sentry
Captures and analyzes errors with stack traces and user context.

DataDog
Provides logging and monitoring across services, with customizable dashboards.

LogDNA
Aggregates logs and supports Pino integration for structured log analysis.

Here's how to integrate Sentry with Fastify for additional error tracking:

```
const Sentry = require('@sentry/node');

Sentry.init({
  dsn: process.env.SENTRY_DSN,
  tracesSampleRate: 1.0, // Capture all transactions
});

fastify.setErrorHandler((error, request, reply) => {
  Sentry.captureException(error);
  request.log.error(error);
  reply.status(500).send({ error: 'An internal error occurred' });
});
```



3.12 TypeScript

TypeScript is a superset of JavaScript that brings optional static typing, enhancing code maintainability, readability, and robustness.

For complex enterprise APIs, TypeScript helps avoid runtime errors by catching issues at compile time and providing clear interfaces, reducing ambiguities that are common in large codebases.

Benefits of Using TypeScript in API Development

Using TypeScript in Fastify applications brings several advantages:

Compile-Time Error Checking
TypeScript highlights type errors before runtime, leading to fewer bugs in production.

Code Autocompletion and IntelliSense
Enhanced editor support offers better developer experience and speeds up development.

Improved Code Readability
Explicit types make code more readable and understandable for other developers.

Reduced Runtime Errors
By defining data structures and types, TypeScript can help avoid issues caused by unexpected data shapes or values.

Setting Up TypeScript with Fastify

To begin using TypeScript in your Fastify application, you'll need to install the necessary dependencies and configure TypeScript for the project:

1. Install Dependencies:

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

2. Initialize TypeScript: Create a `tsconfig.json` in the project root:

```
{
  “extends”: “@fastify/tsconfig”,
  “compilerOptions”: {
      “outDir”: “dist”,
      “sourceMap”: true
  },
  “include”: [“src/**/*.ts”]
}
```

3. Organize Your Project Structure:

- `src`: Place all TypeScript files here.
- `dist`: Compiled JavaScript files will be output here.

4. Configure Fastify: Update Fastify entry points to support TypeScript:

```
import fastify from ‘fastify’;

const app = fastify({ logger: true });

try {
  await app.listen({
      port: 3000,
  });
} catch (err) {
  app.log.error(err);
  process.exit(1);
}01
```

Defining Types for Requests and Responses

TypeScript allows us to define clear types for request parameters, query strings, request bodies, and responses. Here's how to define types for a route:

```
import { FastifyRequest, FastifyReply } from 'fastify';

interface GetUserRequest {
  Params: {
    userId: string;
  };
}

app.get<GetUserRequest>('/users/:userId', async (request, reply) => {
  const { userId } = request.params;
  // Retrieve and return user data
  return { userId };
});
```

Using these types ensures that any mismatches in request structure or data type are caught during development.

Creating Reusable Interfaces and Type Definitions

When building APIs, it's common to have shared data structures (e.g., User, Product). Use TypeScript interfaces and type aliases to define these structures and reuse them across the project:

```
interface User {
  id: string;
  name: string;
  email: string;
  createdAt: Date;
}
```

These interfaces can be shared across modules, ensuring consistency and reusability.

3.13 Type-Safe Configuration and Environment Variables

Managing environment variables with type safety enhances reliability. To ensure type safety, define the expected configuration using TypeScript:

1. Create a Config Interface:

```
interface Config {
  PORT: number;
  DATABASE_URL: string;
  NODE_ENV: 'development' | 'production' | 'test';
}

const config: Config = {
  PORT: parseInt(process.env.PORT || '3000', 10),
  DATABASE_URL: process.env.DATABASE_URL || '',
  NODE_ENV: process.env.NODE_ENV as 'development' | 'production' | 'test'
};
```

2. Validate the Configuration:
   Use a library like zod to validate the configuration values at runtime.

3.14 Handling Errors with Type Safety

Error handling in TypeScript offers the advantage of defining custom error types that can be caught and managed systematically. Define a custom error type to handle known cases:

```
class NotFoundError extends Error {
  constructor(message: string) {
    super(message);
    this.name = 'NotFoundError';
  }
}

try {
  // logic
} catch (error) {
  if (error instanceof NotFoundError) {
    reply.status(404).send({ message: error.message });
  }
}
```

Custom error types make your error handling more structured and meaningful.

3.15 Type-Driven Validation with Typebox

Typebox enables schema definitions that work well with TypeScript, offering runtime validation and static type generation. Here's how to integrate it:

1. Define the Schema:

```
import { Type, Static } from '@sinclair/typebox';

const UserSchema = Type.Object({
  id: Type.String(),
  name: Type.String(),
  age: Type.Optional(Type.Number()),
});

type User = Static<typeof UserSchema>;
```

2. Use Schema for Validation:
   Use Typebox schemas directly in Fastify route validation, ensuring consistency in data structure:

```
app.post<{ Body: User }>('/users', {
  schema: {
    body: UserSchema
  }
}, async (request, reply) => {
  // request.body is already type-checked as User
});
```

Best Practices for Maintaining Type Safety

Use Strict Mode
Enable `strict` mode in `tsconfig.json` to enforce type safety.

Define Clear Interfaces
Structure common interfaces for consistent types.

Leverage Typebox
Use Typebox for both runtime validation and TypeScript typing.

Avoid “any”
Avoid the “**any**” type to maintain type safety and rely on **unknown** if type is unclear.

Type Enforced Dependencies
Ensure all dependencies are typed with **`@types`** packages, or create custom types if necessary.

Testing Route Handlers with JSON schema validation

When defining schemas using TypeBox, you’re actually creating JSON Schemas that Fastify uses to validate incoming data with Ajv at runtime.

Testing your route handlers ensures that these JSON Schema validations work as expected—rejecting incorrect input and allowing valid data.

1. Define the Schema and Route

```
import { Type } from '@sinclair/typebox';
import Fastify from 'fastify';
import { TypeBoxTypeProvider } from '@fastify/type-provider-typebox';

export function buildFastify() {
  const fastify = Fastify().withTypeProvider<TypeBoxTypeProvider>();

  const UserSchema = Type.Object({
    id: Type.String(),
    name: Type.String(),
    age: Type.Optional(Type.Number({ minimum: 0 }))
  });

  fastify.post('/user', {
    schema: {
      body: UserSchema
    }
  }, async (request, reply) => {
    return {
      message: `User ${request.body.name} created.`
    };
  });
}
```

Now, when you need to run your server or test your routes, you can import and invoke `buildFastify` to get a new instance:

```
import { buildFastify } from './server';

const fastify = buildFastify();

fastify.listen({ port: 3000 }, (err, address) => {
  if (err) throw err;
  console.log(`Server running on ${address}`);
});
```

2. Write Tests for Valid and Invalid Data:

```
import test from 'node:test';
import assert from 'node:assert';

test('POST /user - valid data passes', async (t) => {
  const response = await fastify.inject({
    method: 'POST',
    url: '/user',
    payload: { id: '123', name: 'Alice' }
  });

  assert.equal(response.statusCode, 200);
});

test('POST /user - invalid data fails validation', async (t) => {
  const response = await fastify.inject({
    method: 'POST',
    url: '/user',
    payload: { id: '123', name: 123 }
  });

  assert.equal(response.statusCode, 400); // Expect 400 Bad Request
});
```

Here, Typebox ensures that incorrect data (like a numeric `name` value) triggers a `400 Bad Request`. This test confirms that data validation works as expected, even in edge cases.

Integration Testing with Fastify and `node:test`

In an enterprise application, API routes often interact with databases or external services. Integration tests validate that these components work together as expected.

1. Mocking External Services for Integration Tests

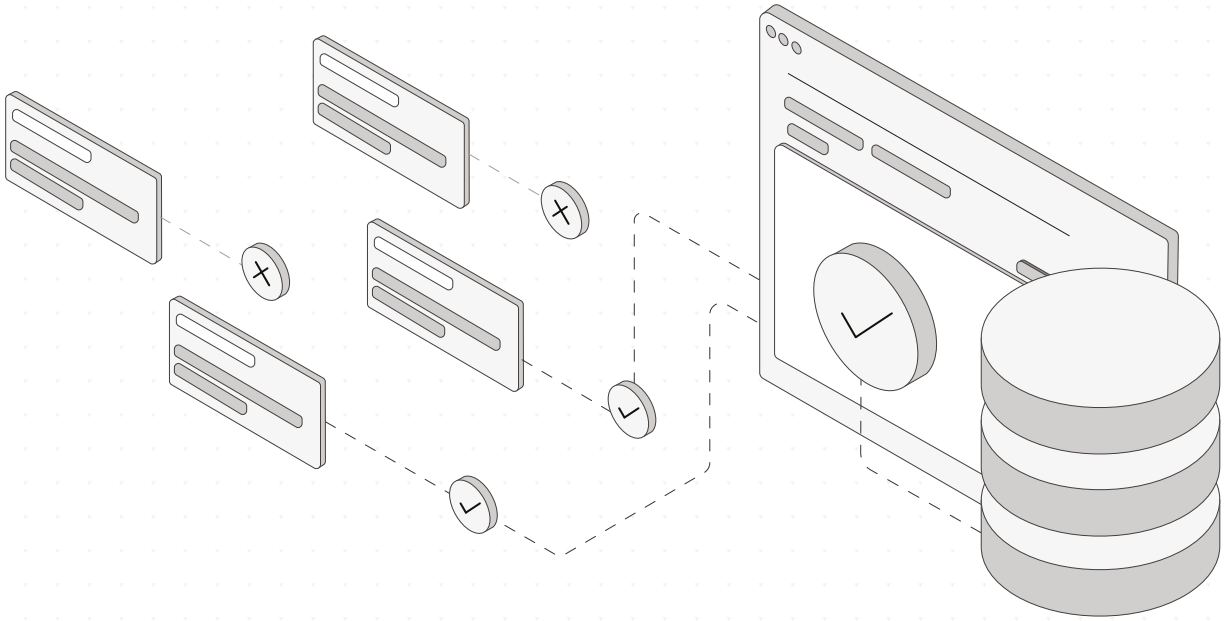
Suppose your Fastify route fetches user data from an external API:

```
fastify.get('/external-data', async () => {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  return data;
});
```

Use a mock to simulate the external API response in your test:

```
import test from "node:test";
import assert from "node:assert";
import { MockAgent, setGlobalDispatcher } from "undici";
import { buildServer } from "../src/server.js";

test("GET /external-data - returns data from external API", async (t) => {
  const fastify = buildServer();

  const mockAgent = new MockAgent();
  setGlobalDispatcher(mockAgent);
  mockAgent.enableNetConnect();

  const mockPool = mockAgent.get("https://apiexample.com");
  mockPool
    .intercept({
      path: "/data",
      method: "GET",
    })
    .reply(200, { id: "abc", value: "Test Data" });

  const response = await fastify.inject({
    method: "GET",
    url: "/external-data",
  });

  assert.equal(response.statusCode, 200);
  assert.deepEqual(await response.json(), { id: "abc", value: "Test
  Data" });

  await fastify.close();
});
```

Using `undici`, you can intercept and simulate the external API's response, ensuring reliable integration tests without actual network requests.



3.16 Introduction to Data validation

Data Validation with Ajv

In API development, ensuring that the incoming data is structured and valid is essential for security, reliability, and maintainability. Ajv is a fast JSON Schema validator that provides robust runtime validation of data. Leveraging JSON Schema offers several key advantages:

Standards Compliance & Interoperability
JSON Schema is widely adopted and closely tied to standards like OpenAPI. This association not only facilitates automated API documentation but also enables client code generation, ensuring consistency between server-side validations and client-side expectations.

Comprehensive Validation
Ajv can enforce a broad range of constraints—types, formats, and custom validations—ensuring that every piece of data entering your system conforms to predefined rules. This reduces the risks of security vulnerabilities such as injection attacks and minimizes the chance of data corruption.

Performance
Built for speed, Ajv provides high-performance validation even under demanding workloads, which is crucial for scalable, enterprise-level systems.

By validating incoming data with Ajv, you establish a robust first line of defense, ensuring that your API handles only well-structured and expected data formats.

Type Safety with TypeBox

While Ajv ensures runtime data integrity, TypeBox complements this by providing static type safety during development. TypeBox allows you to define JSON Schemas using TypeScript syntax, which brings several advantages:

Type Providers
With TypeBox, you can generate schemas that serve as both runtime validators and compile-time type definitions. This dual role helps prevent mismatches between the types your application expects and the validation logic.

Improved Developer Experience
By writing your schemas in TypeScript, you gain immediate feedback from the compiler. This reduces errors early in the development process and improves overall code quality.

Synchronization of Schema and Types
Instead of maintaining separate definitions for runtime validation and static typing, TypeBox ensures that your data validation rules and TypeScript types are always in sync, reducing duplication and the chance of inconsistency.

Using TypeBox together with Ajv creates a powerful combination: robust runtime validation through JSON Schema standards and seamless type safety with TypeScript.

This integrated approach ensures that your API not only enforces data integrity at runtime but also benefits from enhanced developer productivity and code reliability during development.

An alternative to TypeBox is fastify-type-provider-json-schema-to-ts. This package allows Fastify to infer TypeScript types directly from JSON Schemas, eliminating the need for a separate schema-to-type conversion.

It integrates seamlessly with Fastify's validation system, making it a lightweight and flexible option for developers who prefer working with raw JSON Schema while still enjoying TypeScript type inference.

Why Data Validation is Critical in Enterprise Applications

For enterprise applications, data validation is more than just preventing bad inputs; it is a foundational element for:

Ensuring Consistency
Large-scale applications often integrate multiple services, databases, and external clients. Validation ensures that the data exchanged between these systems remains consistent and predictable.

Enhancing Security
Strong data validation limits the attack surface by ensuring that only data conforming to predefined schemas enters your system, preventing exploits like injection attacks.

Improving Developer Efficiency
By catching invalid data early at the API boundary, validation helps reduce bugs that would otherwise surface deeper within your code, saving debugging time and effort.

Ensuring API Contract Fidelity
In enterprise systems, breaking an API contract—when data does not meet the agreed-upon structure—can lead to serious issues for downstream clients or microservices. With strict validation, you ensure API contracts are enforced.

What is Typebox?

Typebox is a powerful library for defining and validating JSON schemas in TypeScript. It helps developers define complex data structures, such as objects, arrays, and nested objects, in a way that can be validated at runtime and statically checked during development.

Here are some key features of Typebox:

Type-Safe Validation
Typebox allows developers to define JSON schemas while also automatically inferring TypeScript types from those schemas.

Fastify Integration
It seamlessly integrates with Fastify's built-in validation mechanisms, enabling you to apply schema validation to your API routes.

Schema Composition
Typebox allows you to create complex schemas using basic building blocks like `Type.Object`, `Type.String`, and `Type.Array`.

Defining Schemas with Typebox

A schema defines the structure and rules for the data your API expects. Typebox provides several built-in types that allow you to construct these schemas easily. Below are some examples:

```
import { Type } from '@sinclair/typebox';

// Defining a simple schema for a user object
const UserSchema = Type.Object({
  // name must be a non-empty string
  name: Type.String({ minLength: 1 }),
  // age is optional, must be a non-negative number
  age: Type.Optional(Type.Number({ minimum: 0 }))
});

// You can create more complex schemas with nested objects or arrays
const ProductSchema = Type.Object({
  title: Type.String(),
  price: Type.Number({ minimum: 0 }),
  tags: Type.Array(Type.String()),  // array of strings for tags
});
```

Here, `Type.Object` creates a schema for objects, while `Type.String()` and `Type.Number()` ensure that fields match the specified types. The schema also allows additional options like `minLength` or `minimum` to enforce further validation rules.

Validating API Requests

Once you've defined your schemas, integrating them with Fastify's route validation is simple. Fastify uses these schemas to validate incoming requests, ensuring that the data adheres to the rules you've set up.

For example, if you want to validate the payload of a POST /users request, you can do so like this:

```
import Fastify from 'fastify';
import { Type } from '@sinclair/typebox';
import { TypeBoxTypeProvider } from '@fastify/type-provider-typebox';

const fastify = Fastify();

// Create a Fastify instance with the TypeBox type provider const
fastify = Fastify().withTypeProvider<TypeBoxTypeProvider>();

// Define the schema for user creation
const userSchema = Type.Object({
  name: Type.String(),
  age: Type.Number({ minimum: 0 })
});

// Register a route with schema validation
fastify.post('/users', {
  schema: {
    body: userSchema  // Validate the request body against userSchema
  }
}, async (request, reply) => {
  const { name, age } = request.body;
  // Process user creation
  reply.send({ message: `User ${name} is ${age} years old.` });
});

fastify.listen({ port: 3000 }, (err) => {
  if (err) throw err;
  console.log('Server running on http://localhost:3000');
});
```

If a client sends invalid data to this route, Fastify will automatically return a `400 Bad Request` response, ensuring that only valid data is processed by your application.

Type Inference with TypeScript

One of Typebox's most valuable features is its ability to infer TypeScript types from the schemas you define.

This allows you to enjoy the benefits of both runtime validation and compile-time type safety:

```
const UserSchema = Type.Object({
  name: Type.String(),
  age: Type.Optional(Type.Number()),
});

// Infer TypeScript type from schema
type User = Static<typeof UserSchema>;

// Now you can use the 'User' type throughout your code
function createUser(user: User) {
  console.log(`Creating user: ${user.name}`);
}
```

The `Static` utility from Typebox automatically generates TypeScript types based on your schema, giving you strict type-checking while writing your application.

Handling Validation Errors

Fastify not only validates data but also provides helpful error responses. For example, if a required field is missing or a type mismatch occurs, Fastify sends a detailed error message back to the client. Here's how you can customize error responses:

```
fastify.setErrorHandler((error, request, reply) => {
  if (error.validation) {
    reply.status(400).send({ error: 'Invalid request data', details:
error.validation });
  } else {
    reply.status(500).send({ error: 'Internal Server Error' });
  }
});
```

This way, you can ensure that clients receive meaningful feedback when they provide invalid data, improving the overall API experience.

Best Practices for Data Validation in Enterprise Systems

As your application scales, so does the complexity of your data validation requirements. Here are a few best practices to consider:

Handling Optional and Required Fields

In enterprise systems, schemas can evolve over time. Clearly document which fields are required and which are optional to avoid breaking clients relying on older versions of your API.

```
const schema = {
  body: {
    type: 'object',
    properties: {
      username: {
        type: 'string'
      }, // Required field
      middleName: {
        type: 'string'
      }, // Optional field
      phoneNumber: {
        type: ['string', 'null']
      }, // Optional, can be string or null
      role: {
        type: 'string',
        default: 'user'
      }, // Optional, defaults to "user"
      age: {
        type: ['number', 'null']
      } // Optional, can be number or null
    },
    required: ['username'] // Only "username" is mandatory
  }
};
```

Automated Tests for Validation Logic

Write tests that ensure your validation logic works as expected, especially when introducing new schema rules.

Versioning Schemas

When updating API schemas, ensure backward compatibility by versioning your APIs or allowing legacy schemas.

Wrapping Up

Building robust, scalable, and high-performance APIs with Fastify involves much more than just setting up routes. With its extensive plugin ecosystem, support for TypeScript, powerful validation capabilities via TypeBox and Ajv, and built-in testing tools, Fastify makes it easier to craft modern, enterprise-grade applications.

The integration with tools like Undici and the ease of connecting to various databases, alongside best practices for graceful shutdowns, error handling, and configuration management, ensures that your APIs are not only fast but also reliable and secure.

Whether you're managing data flow, handling high concurrency, or preparing your app for the challenges of real-world traffic, Fastify provides the foundation for building efficient, maintainable, and performant APIs.

By leveraging the right testing strategies and performance optimisations, you can ensure your Fastify APIs will scale to meet the demands of enterprise environments with confidence.

At Platformatic, we believe that Open Source projects like Fastify are the backbone of innovation. As a proud sponsor and active contributor to the Fastify project, Platformatic is able to give back directly to the tools we rely on, ensuring that they remain sustainable, secure, and future-proof.

These projects are critical to the developer community and the broader tech industry, enabling teams to build and scale applications with greater efficiency. Platformatic itself was born out of the desire for us to build upon the technical foundations of Fastify.

Today, Platformatic seamlessly integrates with the Fastify web framework and extends its capabilities while incorporating all the expertise in building, operating, and scaling Node.js applications that we have accumulated over the past 10 years from the Open Source Community.

04

Building SSR Frontends

SSR frameworks, a guide to building a basic SSR page, deployment considerations, and more

04

Building SSR Frontends

SSR frameworks, a guide to building a basic SSR page, deployment considerations, and more

Server-Side Rendering (SSR) is a web development technique where HTML content is generated on the server and sent to the client. Unlike client-side rendering, where JavaScript runs in the browser to generate content, SSR ensures that fully-rendered HTML is delivered to the user's browser upon the initial request.

One of the primary benefits of SSR is its impact on Search Engine Optimization (SEO). Search engine crawlers, which often struggle with JavaScript-heavy client-side applications, can easily parse and index the pre-rendered HTML provided by SSR.

Additionally, SSR improves initial load times since the browser does not need to wait for JavaScript to execute before displaying meaningful content. This, in turn, leads to an enhanced user experience, particularly for users on slow networks or devices with limited processing power.

Node.js is particularly well-suited for SSR due to its ability to handle server-side tasks efficiently. With its asynchronous, event-driven architecture and robust JavaScript runtime, Node.js excels at fetching data from APIs and dynamically generating HTML content. It also integrates seamlessly with frameworks like React, making it a strong choice for developers looking to implement SSR in modern web applications.

By combining the strengths of Node.js and SSR, developers can create applications that are not only performant but also deliver content-rich experiences to users across various environments, reinforcing the importance of SSR in today's web ecosystem.

However, building an SSR system comes with its challenges.

One of the primary concerns is resource intensity—SSR can be incredibly demanding on server resources, as it requires rendering pages on the server for each request, leading to increased CPU and memory usage.

Additionally, developers must consider caching strategies, load balancing, and efficient data fetching to ensure scalability and performance. Proper error handling and fallback mechanisms are also critical, as server-side rendering can introduce complexities in managing client-server interactions.

Despite these challenges, when implemented thoughtfully, SSR can significantly enhance user experience and search engine optimization (SEO), making it a valuable tool in modern web development.

Frameworks for SSR with React

To simplify the implementation of SSR, several frameworks built on top of Node.js have emerged.

These frameworks provide tools and abstractions that streamline the development process, allowing developers to focus on building features rather than configuring SSR from scratch.

Next.js: A Full-Fledged Framework

Next.js is one of the most popular frameworks for SSR with React according to the State of JavaScript 2024.

It provides a comprehensive solution for building React applications, complete with built-in routing, API routes, and automatic code splitting. Next.js simplifies SSR by enabling developers to define server-rendered pages through its file-based routing system.

Each file in the pages directory corresponds to a route, and developers can use Next.js's getServerSideProps function to fetch data and render it server-side.

Key benefits of using Next.js include:

Ease of setup
Next.js handles most of the heavy lifting, such as bundler configuration and optimization.

Static site generation (SSG)
In addition to Server Side Rendering (SSR), Next.js supports static generation, offering a hybrid approach to rendering. With Incremental Static Regeneration (ISR), developers can update static content after deployment, ensuring a balance between the performance benefits of SSG and the flexibility of SSR for dynamic updates.

API routes
Developers can create serverless API endpoints directly within the application, which can be convenient for small-scale projects or prototypes. However, for larger applications, dedicated API frameworks are often a better choice, as they provide robust features like validation, OpenAPI support, and enhanced scalability.

React with Vite: A lightweight addition

While Next.js provides a full-fledged framework, **Vite** is a lightweight build tool designed for speed and simplicity. It excels at fast development builds and offers some SSR capabilities when combined with additional libraries.

Unlike Next.js, Vite focuses on the developer experience, offering an unopinionated approach that allows developers to piece together the tools they need for their specific use case.

Key differences between Next.js and Vite include:

Philosophy
Next.js is opinionated and includes a comprehensive set of tools for building SSR applications, while Vite emphasizes flexibility and performance as a build tool.

Setup complexity
Vite requires additional configuration and integration with libraries to achieve SSR, whereas Next.js provides a more out-of-the-box solution.

For most projects prioritizing SSR, Next.js stands out as the go-to choice, offering a robust ecosystem and extensive documentation to accelerate development.

Setting Up a Demo Project

In this chapter, we'll create an event listing application using Next.js 13+ with App Router.

This project will demonstrate server-side rendering (SSR), dynamic routing, and modern React features like Streaming and Suspense.

Project Overview

Our demo project is an events platform that allows users to:

- View a list of upcoming events
- See detailed information about each event
- Edit event details
- Navigate between pages seamlessly

This practical example will showcase real-world implementation of Next.js features while building something useful.

Initial Setup

First, create a new Next.js project with the following configuration:

```
npx create-next-app@15 event-listing-app --javascript --tailwind --app
```

This command sets up:

- Next.js 15 with App Router
- JavaScript configuration
- Tailwind CSS for styling
- Modern directory structure

This practical example will showcase real-world implementation of Next.js features while building something useful.

Project Structure

The project follows this organization:

```
event - listing - app /
├── .env.local                # Environment variables
├── app /
│   ├── layout.js             # Root layout
│   ├── page.js               # Home page
│   ├── events /
│   │   ├── page.js           # Events listing page
│   │   └──[id] /
│   │       └── page.js       # Individual event page
│   ├── api /
│   │   └── events /
│   │       └── route.js      # API route for events
│   ├── lib /
│   │   └── api /
│   │       └── events.js     # API client
│   └── components /
│       ├── ui /              # Reusable UI components
│       └── events /          # Event - specific components
```

Dependencies

Install additional dependencies needed for the project:

```
npm install date-fns clsx @heroicons/react
```

These packages provide:

- `date-fns:` Date formatting and manipulation
- `clsx:` Conditional class name construction
- `@heroicons/react:` Icon components

Initial Component Setup

Let's create our first component to verify the setup. Replace the content of `app/page.js` with:

```
import Link from 'next/link';
export default function Home() {
  return (
    <main className="container mx-auto px-4 py-8">
      <h1 className="text-4xl font-bold mb-8">
        Welcome to Event Listing
      </h1>
      <Link href ="/events"
        className ="bg-blue-600 text-white px-6 py-3 rounded-lg
        hover:bg-blue-700">
        Browse Events
      </Link>
    </main>
  );
}
```

Verifying the Setup

Start the development server:

```
npm run dev
```

Visit `http://localhost:3000` to verify:

- The application loads correctly
- Tailwind styles are working
- The basic routing functions

4.2 Building a Basic SSR Page

Now, we'll create our first server-side rendered page using Next.js. We'll build an events listing page that will showcase the power of SSR for improved performance and SEO.

You can find all code referenced in this book in the following Github repository: https://github.com/platformatic/events-app-ebook

---

Understanding Server-Side Rendering

Before diving into the code, let's understand why SSR is beneficial for our events platform example. It offers:

- Better SEO as search engines can crawl the fully rendered content
- Faster initial page load and First Contentful Paint (FCP)
- Improved performance on slower devices
- Better social media sharing with proper meta tags

Creating the Events List Page

In this section, we'll build an SSR-powered page to display a list of events in `app/events/page.js`.

1. Setting Up Metadata and file imports

```
import { getEvents } from "@/app/lib/api/events";
import EventList from "@/app/components/events/event-list";
import { Suspense } from "react";

export const metadata = {
  title: "Events | Event Listing App",
  description: "Discover upcoming events in your area",
};
```

- `title:` Specifies the title that appears in the browser tab and improves SEO.
- `description:` Provides a meta description for search engines, enhancing discoverability.

2. Defining the SSR Page Component

The `EventsPage` component is the default export for this file. It's marked as `async` to enable server-side data fetching for all upcoming events.

```
export default async function EventsPage({ searchParams }) {
  const page = Number(searchParams.page) || 1
  const events = await getEvents(page)
  return (
    <main className="container mx-auto px-4 py-8">
      <h1 className="text-3xl font-bold mb-8">Upcoming Events</h1>
      <Suspense fallback={<EventListSkeleton />}>
        <EventList events={events} />
      </Suspense>
    </main>
  )
}
```

Building the Event Card Component

Create `app/components/events/event-card.js:`

```
import Image from 'next/image';
import Link from 'next/link';
import { format } from 'date-fns';
```

- `Image` (from `next/image`): Used to efficiently handle image optimization, lazy loading, and responsive image sizes.
- `Link` (from `next/link`): Enables client-side navigation to other pages within the Next.js app.
- `format` (from `date-fns`): Formats the event's start date into a comprehensible format.

```
export default function EventCard({ event }) {
  return (
    <Link
      href={`/events/${event.id}`}
      className="group block overflow-hidden rounded-lg border bg-
      white shadow-sm"
    >
      <div className="relative h-48 w-full">
        <Image
          src={event.imageUrl || '/images/placeholder.jpg'}
          alt={event.title}
          fill
          className="object-cover"
          sizes="(max-width: 768px) 100vw, (max-width: 1200px) 50vw,
          33vw"
        />
      </div>

      <div className="p-4">
        <h3 className="text-lg font-semibold group-hover:text-
        blue-600">
          {event.title}
        </h3>

        <div className="mt-2 text-sm text-gray-600">
          <p>{format(new Date(event.startDate), 'PPP')}</p>
          <p>{event.location}</p>
        </div>
      </div>
    </Link>
  );
}
```

Here, we are lazy-loading an image using the `next/image`. If no image is provided, a fallback placeholder is used.

The `fill` property ensures the image fully covers its container, while the `sizes` attribute optimizes loading based on screen width.

The event title and details, including the formatted event date and location, are displayed inside a styled container.

Server-Side Data Fetching

Create `app/lib/api/events.js`:

```
export async function getEvents(page = 1) {
  try {
    const response = await fetch(
      `${process.env.NEXT_PUBLIC_API_URL}/db/events/?page=${page}`,
      {
        next: {
          revalidate: 60
        }
      }
    )
    if (!response.ok) {
      throw new Error(
        `Failed to fetch events: ${response.statusText}`
      )
    }
    return {
      data: await response.json(),
      error: null
    }
  } catch (error) {
    console.error(`Error fetching events:`, error)
    return {
      data: null,
      error: `Failed to load events.Please try again later.`
    }
  }
}
```

Loading State with Suspense

Suspense is a React feature that lets you display a fallback until your children have finished loading for user interfaces.

Create `app/components/loading/event-card-skeleton.js`, and inside this file, we will create a placeholder UI that displays while fetching data from the API.

```
export function EventListSkeleton() {
  return (
    <div className= “grid grid-cols-1 gap-6 md:grid-cols-2 lg:grid-
    cols-3” >
      { [1, 2, 3].map((n) => (
        <div key= { n } className = “animate-pulse” >
          <div className=”h-48 bg-gray-200 rounded-t-lg” />
          <div className=”p-4 space-y-3” >
            <div className=”h-6 bg-gray-200 rounded w-3/4” />
            <div className=”h-4 bg-gray-200 rounded w-1/2” />
          </div>
        </div>
      ))}
    </div>
  );
}
```

Benefits of This Implementation

Our SSR page provides several advantages:

SEO Optimization

- Content is rendered on the server
- Search engines receive complete HTML
- Meta tags are properly set

Performance

- Fast initial page load
- Minimal client-side JavaScript
- Efficient image loading with next/image

User Experience

- No content flashing
- Smooth loading states
- Progressive enhancement

Maintainability

- Clean component structure
- Separated concerns
- Reusable components

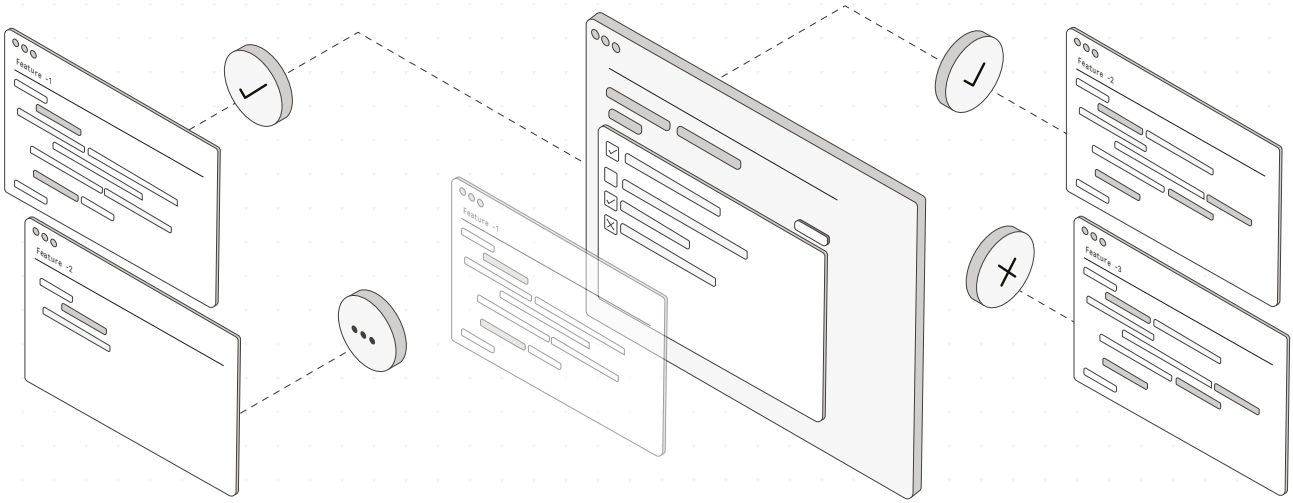
Here are some performance downsides of using Suspense with SSR:

Increased Server Load

- Each request requires server-side rendering
- Higher server CPU and memory usage
- Potential bottlenecks under heavy traffic

Network Impact

- Larger initial HTML payload
- Multiple roundtrips for data fetching
- Additional bandwidth usage for loading states



4.3 Adding Features

Now, we'll enhance our events platform with additional features that leverage Next.js' powerful capabilities. We'll implement dynamic routing, optimize performance, and add modern React features.

Implementing Dynamic Routing

Next.js provides automatic routing based on the file system. Let's create a dynamic route for individual event pages.

Create `app/events/[id]/page.js` file and inside it:

```
import { Suspense } from 'react';
import { getEventById } from '@/app/lib/api/events';
import EventDetail from '@/app/components/events/event - detail';
import { EventDetailSkeleton } from '@/app/components/loading/event -
card - skeleton';

export default async function EventPage({ params }) {
  const event = await getEventById(params.id);

  return (
    <Suspense fallback= {<EventDetailSkeleton />}>
      <EventDetail event={ event } />
        </Suspense>
  );
}
```

Dynamic Data Fetching

Update `app/lib/api/events.js` to handle parameter-based fetching:

```
export async function getEventById(id) {
  try {
    const response = await fetch(
      `${process.env.NEXT_PUBLIC_API_URL}/db/events/${id}`,
      { next: { revalidate: 60 } }
    );

    if (!response.ok) {
      throw new Error('Event not found');
    }

    return response.json();
  } catch (error) {
    console.error('Error fetching event:', error);
    throw error;
  }
}
```

Image Optimization

Next.js provides built-in image optimization. Let's implement it in our EventDetail component:

```
import Image from 'next/image';
export default function EventDetail({ event }) {
  return (
    <article className="max-w-4xl mx-auto">
      <div className="relative h-96 rounded-lg overflow-hidden">
        <Image
          src={event.imageUrl}
          alt={event.title}
          fill
          className="object-cover"
          priority
          sizes="(max-width: 1024px) 100vw, 1024px"
          quality={90}
        />
      </div>
      {/* Rest of the component */}
    </article>
  );
}
```

Implementing Prefetching

Add prefetching to improve navigation performance:

```
import Link from 'next/link';

export default function EventCard({ event }) {
  return (
    <Link
      href={`/events/${event.id}`}
      prefetch={true}
      className="group block..."
    >
      {/* Card content */}
    </Link>
  );
}
```

Streaming and Suspense Implementation

Let's implement streaming for progressive loading in the `app/events/page.js` file:

```
import { Suspense } from 'react';

async function EventsContent({ page }) {
  const events = await getEvents(page);
  return <EventList events = { events } />;
}

export default function EventsPage({ searchParams }) {
  return (
    <main className="container mx-auto px-4 py-8">
      <h1 className="text-3xl font-bold mb-8">Upcoming Events</h1>
      <Suspense fallback={<EventListSkeleton />}>
        <EventsContent page={ searchParams.page } />
       </Suspense>
    </main>
  );
}
```

Create loading UI components:

```
// app/components/loading/loading-states.js
export function EventListSkeleton() {
  return (
    <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3
    gap-6">
      {[1, 2, 3].map((n) => (
        <div key={n} className="animate-pulse">
          <div className="h-48 bg-gray-200 rounded-t-lg" />
          <div className="p-4 space-y-3">
            <div className="h-6 bg-gray-200 rounded w-3/4" />
            <div className="h-4 bg-gray-200 rounded w-1/2" />
          </div>
        </div>
      ))}
    </div>
  );
}
```

Error Handling

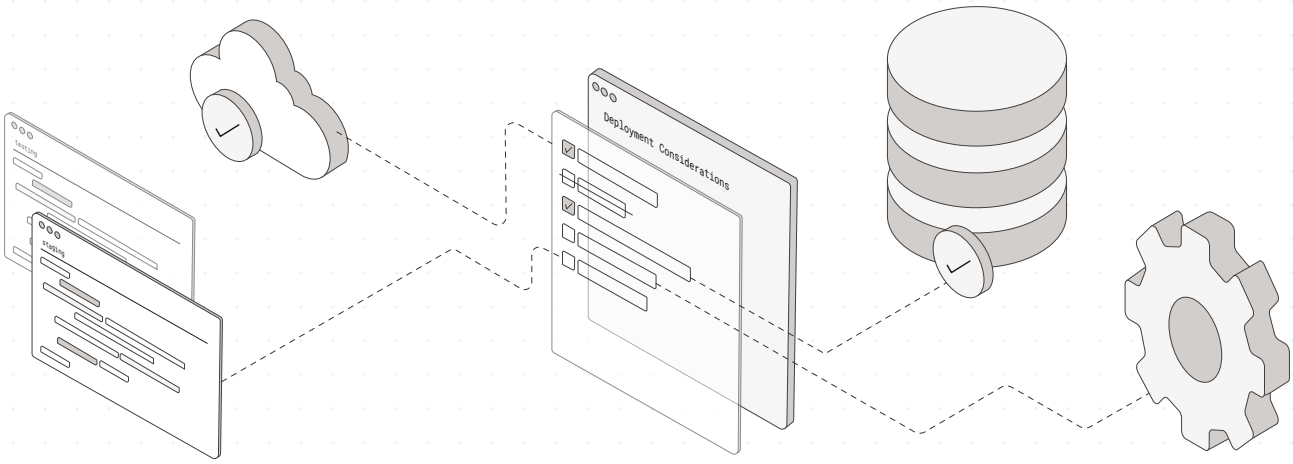
Implement error boundaries for better error handling:

```
// app/error.js
'use client';
export default function Error({ error, reset }) {
return (
  <div className="text-center py-12" >
    <h2 className="text-2xl font-bold text-red-600 mb-4" >
      Something went wrong!
    </h2>
    <button onClick={() => reset() }
      className="bg-blue-600 text-white px-4 py-2 rounded">
      Try again
    </button>
  </div>
)
```

Performance Optimizations

Add metadata for better SEO:

```
// app/events/[id]/page.js
export async function generateMetadata({ params }) {
  const event = await getEventById(params.id);

  return {
    title: `${event.title} | Event Listing App`,
    description: event.description.slice(0, 160),
    openGraph: {
      title: event.title,
      description: event.description,
      images: [event.imageUrl],
    },
  };
}
```



4.4 Deployment Considerations

Now let's explore different deployment strategies for both Next.js and Vite-based React applications, along with essential caching considerations for optimal performance.

A Note on Incremental Static Regeneration (ISR)

When using Next.js ISR in enterprise environments, one common concern is that **Next.js may perform fetch calls during the build**. This can create environment-specific artifacts if you're promoting the same build output from, say, staging to production.

Why is this an issue?

If your build fetches data from environment-specific endpoints, the generated static files will be "locked" to data from that environment.

Consequently, promoting the same artifact to a different environment without re-building can cause mismatched data, broken links, or other inconsistencies.

How to handle it?

- Disable or limit ISR calls at build time if you plan to re-use the same bundle across multiple environments.
- Use runtime environment variables for data fetching instead of doing fetch calls at build time.
- Configure revalidation logic to happen at runtime, so your app can update data after deployment without a rebuild.

Deployment Options for Next.js

1. Vercel (Recommended for Next.js)

Vercel, created by the Next.js team, offers the most streamlined deployment experience:

```
# Install Vercel CLI
npm install -g vercel

# Deploy
vercel
```

Benefits:

- Zero-configuration deployment
- Automatic CI/CD pipeline
- Built-in edge caching
- Serverless functions support
- Automatic HTTPS
- Preview deployments for branches

2. AWS Amplify

AWS Amplify provides a robust platform for Next.js applications:

```
# amplify.yml
version: 1
frontend:
phases:
build:
commands:
- npm ci
  - npm run build
artifacts:
baseDirectory: .next
files:
- '**/*'
cache:
paths:
- node_modules/**/ *
```

Benefits:

- AWS service integration
- Global CDN
- Easy scaling
- Managed database options

3. Traditional Hosting (Self-hosted)

In some cases, you may prefer (or be required) to run Next.js on your own server—whether for on-premises infrastructure, custom routing, compliance, or other reasons. Below is a minimal example of a custom Next.js server:

For custom server setups:

```
// server.js
const { createServer } = require('http')
const { parse } = require('url')
const next = require('next')

const dev = process.env.NODE_ENV !== 'production'
const app = next({ dev })
const handle = app.getRequestHandler()

app.prepare().then(() => {
  createServer((req, res) => {
    const parsedUrl = parse(req.url, true)
    handle(req, res, parsedUrl)
  }).listen(3000)
})
```

Handling ISR and Caching

When using a custom server, certain Next.js features—particularly Incremental Static Regeneration (ISR)—won't automatically work as they do on platforms like Vercel or Amplify. ISR relies on serverless functions, edge caching, and revalidation triggers that must be set up manually or via additional tooling:

Revalidation
You'll need to explicitly route calls to Next.js' revalidation endpoints to regenerate static pages.

Cache Headers
If you're controlling the server layer, you're also responsible for configuring cache headers (`Cache-Control`, `ETag`, etc.) to ensure stale pages are properly revalidated or invalidated.

On-Demand Revalidation
Many teams use a secure API route in Next.js to trigger on-demand revalidation (e.g., when CMS content is updated). Make sure your server forwards those requests properly to Next.js.

OpenNext: Deploying Next.js on Serverless Platforms

OpenNext is a community-driven project that brings Vercel-like serverless capabilities to AWS, Cloudflare, and Netlify.

It enables seamless deployment of Next.js applications while preserving key features such as Incremental Static Regeneration (ISR), on-demand revalidation, and optimized caching strategies.

By providing preconfigured deployment templates, OpenNext eliminates much of the complexity involved in self-hosting Next.js at scale.

Developers looking to deploy Next.js on AWS without major rewrites can leverage OpenNext to achieve serverless scaling, cost efficiency, and better performance—all while maintaining full compatibility with the Next.js framework.

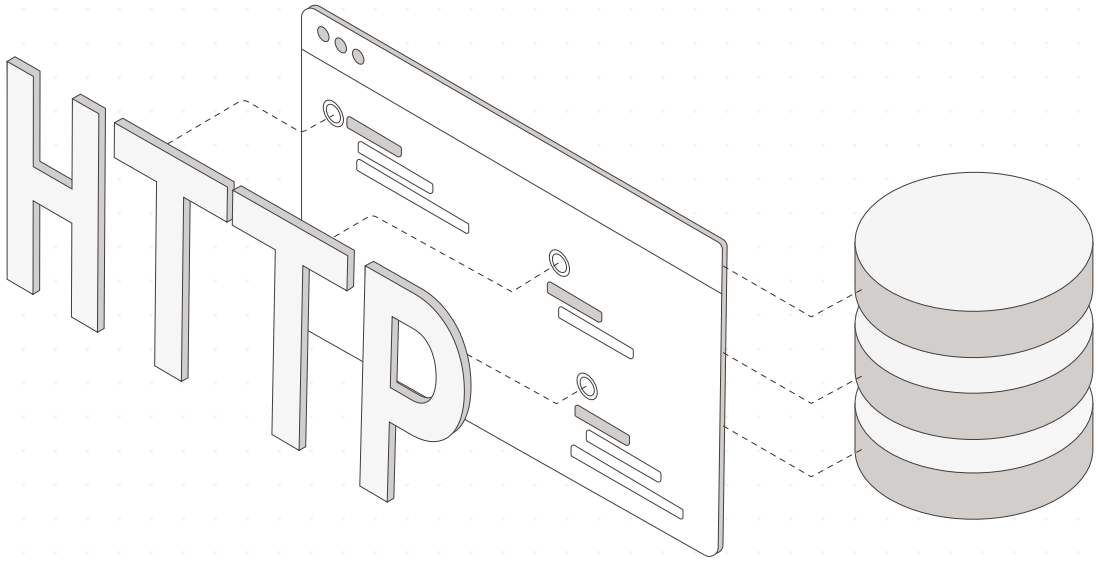
Key Components for Non-Vercel Deployment

To successfully run Next.js outside of Vercel, several critical areas must be addressed:

- Static asset handling and serving
- Server-side rendering for pages and API calls
- Resource caching
- Image optimization
- CDN integration
- Comprehensive routing

When considering OpenNext for deployment, be aware that:

- The implementations are platform-specific and not interchangeable
- Feature support varies significantly between platforms
- Each implementation has its approach to handling core Next.js functionalities
- The AWS implementation requires additional tooling (SST) for deployment



4.5 Understanding HTTP caching

Caching is a technique that allows to improve overall performance by saving cacheable resources to local and fast storage. Subsequent requests can be served using the local stored results rather than requesting a remote resource or performing an expensive computation a second time.

In HTTP, servers specify if a resource can be cached from the clients using the `Cache-Control` header. Additionally, the `ETag`, `Last-Modified` and `Vary` headers provide information about the content and are used by the client to respect the policy specified in the Cache-Control header.

When a client caches a resource, its cache is defined as “private”, if a request is performed from a CDN or another HTTP Proxy instead, that cache is defined as “public” or “shared”.

The `Cache-Control` header defines policies for private and shared caches.

For instance, consider the following response header for a resource:

```
public, max-age=60, s-maxage=10, stale-while-revalidate=59
```

This defines the resources as:

- publicly cacheable (`public`)
- with a lifetime of 1 minute in private caches (`max-age=60`)
- with a lifetime of 10 seconds in shared (public) caches (`s-maxage=10`)
- Additionally valid for another 59 seconds after expiration (stale) if revalidation is performed in background (`` `stale-while-revalidate=59` ``)

The `ETag` header is used to send the client the unique identifier of the resources being returned. When the resource changes its ETag is guaranteed to change. The ETag is generated by the server and its generation technique is unknown and needed by the client.

The `Vary` hearer is used to send the client the request headers, if any, that concurred to the generation of the returned resource. This means that the client should consider the request headers specified in the Vary response header to cache the resource. Different request headers (which are included in the `Vary` header) for the same resource will result in different entries in the client cache.

Next.js Automatic Caching Features

Next.js provides built-in caching optimizations without additional configuration. It automatically optimizes static assets with optimal cache headers, caches static pages at the route level, and enables data caching for fetch requests. Additionally, Next.js includes CDN-compatible caching headers and applies automatic static optimization to eligible pages, ensuring efficient performance and reduced load times.

Configuring Incremental Static Regeneration (ISR) caching

Next.js enables **automatic cache revalidation** based on a specified time interval. You can set the `revalidate` property to define how often the page should be regenerated.

```
export const revalidate = 3600;

async function Page() {
  const response = await fetch('https://api.example.com/data', {
    next: {
      revalidate: 3600
    }
  });

  const data = await response.json();

  return <div> {
    JSON.stringify(data)
  } </div>;
}

export default Page;
export default Page;
```

Here, the page fetches external data and caches it for **one hour (3600 seconds)**. After that, the cache is **automatically invalidated**, and a new request is made in the background while serving the stale content until the updated page is ready.

Client-Side Data Caching

Using libraries like SWR or React Query to cache data in memory:

```
import useSWR from 'swr';

function Profile() {
  const { data } = useSWR('/api/user', {
    revalidateOnFocus: false,
    revalidateOnReconnect: false,
    dedupingInterval: 3_600_000
  });
  return <div>{
    data?.name
  } < /div>;
}
```

Using libraries for caching data in memory has benefits such as instant access from memory, automatic revalidation, and request deduplication.

Browser-Level HTTP Caching

Utilizes browser's built-in HTTP cache through response headers:

```
// next.config.js

module.exports = {
  async headers() {
    return [
      {
        source: '/api/:path*',
        headers: [
          {
            key: 'Cache-Control',
            value: 'private, max-age=3600'
          }
        ]
      }
    ]
  }
}
```

Using the browser approach with Next.js ISR persists across page reloads, works with CDN caching, and reduces network requests.

Static Generation with Incremental Static Regeneration (ISR)

ISR lets you create or update static pages after you build your Next.js application without requiring a full rebuild. This is especially powerful for pages that don't change frequently but still need to be periodically updated

```
// pages/events/[id].js
export async function getStaticProps({ params }) {
  return {
    props: {
      event: await getEvent(params.id)
    },
    revalidate: 60 // Regenerate page every 60 seconds
  }
}
```

Why Use It?

Performance
Pre-renders pages are static files for fast delivery.

Fresh Data
Automatically revalidates and regenerates pages at runtime, meaning you don't need a new build for each content update.

How It Works

Initial Request
When a user visits a statically generated page, Next.js serves a pre-rendered HTML.

Revalidation Timer
After revalidate seconds, the next request triggers Next.js to rebuild that page in the background.

Automatic Update
Subsequent visitors see the updated page instantly once regeneration completes.

Key Considerations

On-Demand Revalidation
You can also trigger re-validation programmatically (e.g., from a CMS webhook) by hitting a special API route.

Caching Behavior
Even though the HTML file is static, if you're self-hosting, you must ensure your server or CDN handles revalidation logic correctly. Platforms like Vercel handle this automatically.

Data Fetching at Build Time vs. Runtime
For truly dynamic data, combine ISR with client-side fetching or serverless APIs.

Client-Side Caching

Client-side caching stores and reuses data within the user's browser or application state. React Query, SWR, or Apollo Client can manage fetch requests, cache responses, and keep data in sync.

Why Use It?

Instant UI Updates
Returns cached data immediately while revalidating in the background.

Reduced Network Requests
Fewer redundant calls to your API when data is already available.

Offline or Low-Connectivity Use
Users can see cached data even if they temporarily lose network access.

```
function useEvents() {
  return useQuery('events', fetchEvents, {
    staleTime: 5 * 60 * 1000, // Consider data fresh for 5 minutes
    cacheTime: 30 * 60 * 1000 // Keep unused data for 30 minutes
  })
}
```

staleTime
The period during which data is considered "fresh." If a user re-requests the data within this window, no new fetch happens.

cacheTime
How long unused data remains in memory or storage before being garbage-collected.

Key Considerations

Choosing a Library
React Query, SWR, and Apollo Client each have caching strategies. Pick the one that fits your data-fetching pattern (REST vs. GraphQL, etc.).

Cache Invalidation
Manually or automatically fetch data when it becomes outdated (e.g. after a user updates an event).

Prefetching
Load data before a component mounts to avoid long loading states.

4.6 Production Checklist to enable caching

Before deploying your Next.js application to production, it's crucial to implement both performance optimizations and security measures.

This checklist covers essential configurations to ensure your application runs efficiently and securely in production.

1. **Enable compression**
   Next.js uses gzip compression by default for static files, but you can enhance this at the application level:

```
// next.config.js
module.exports = {
  compress: true, // Enable gzip compression
  experimental: {
      // Enable modern compression techniques
      compression: true,
      // Enable modern image formats
      modern: true
  }
}
```

2. Enable a custom ISR cache handler for shared caching

Next.js supports custom cache ISR handlers. This is ideal to build shared caches that can be shared between multiple instances of the same application. This will bring massive performance gains since the default implementation is based on the local filesystem, which is not shared between instances.

To use a custom handler, simply provide the file to the configuration file:

```
// next.config.js
module.exports = {
  cacheHandler: require.resolve('./cache-handler.js'),
  cacheMaxMemorySize: 0, // disable default in-memory caching
}
```

The full documentation to build a ISR cache handler can be found in the Next.js documentation. Platformatic provides a Valkey based Next.js cache handler, which can be found here.

3. Image Optimization:

Next.js provides built-in image optimization through the Image component. This automatically handles:

- Resizing images for different devices
- Converting to modern formats like WebP
- Lazy loading for improved performance

```
import Image from 'next/image';

export default function OptimizedImage() {
  return (
    <Image
      src="/large-image.jpg"
      alt="Optimized image"
      width={800}
      height={600}
      placeholder="blur"  // Shows blur effect while loading
      quality={75}        // Balance quality and size
    />
  );
}}
```

4. Assets optimization

Optimize your static assets by configuring proper caching and minimization:

```
/ next.config.js
module.exports = {
  // Enable static asset optimization
  optimizeFonts: true,
  // Configure asset prefix for CDN
  assetPrefix:
    process.env.NODE_ENV === 'production'
      ? 'https://cdn.example.com'
      : '',
  // Minimize CSS
  webpack: (config) => {
    config.optimization.minimizer.push(new CssMinimizerPlugin())
    return config
  }
}
```

5. Use production builds

Create optimized production builds with features like:

- Code splitting
- Package bundling

```
# Next.js production build
npm run build

# Analyze bundle size
ANALYZE=true npm run build
```

4.7 Automatic integration of a Next.js into Watt

As mentioned above, Platformatic provides out-of-the-box a Valkey or Redis based ISR cache.

The integration seamlessly integrates with your Next.js without any modification on your side and allows you to to improve performance of your website by using a fast shared cache for all your pages and the remote resources they might fetch while performing SSR.

To move your existing application to Platformatic, let's transform it into a Watt application.

1. Create a new Watt application.

```
mkdir new-app
cd new-app
npx wattpm@latest init
```

2. Move your existing application inside the `web` folder.

```
mv ../old-app web/app
```

3. Create a `watt.json` file inside `web/app` with the following contents:

```
{
  "$schema": "https://schemas.platformatic.dev/@platformatic/
  next/2.44.0.json",
  "cache": {
    "adapter": "valkey", // "redis" will also work
    "url": "valkey://{CACHE_SERVER_URL}",
    "prefix": "watt-valkey"
  }
}
```

4. Create a `.env` file which sets the `CACHE_SERVER_URL` variable to the actual URL of your Valkey/Redis server. Example:

```
CACHE_SERVER_URL=localhost:6379
```

You are all set! When starting, Watt will automatically instrument your existing Next.js application to use its ISR cache handler.

Security Considerations

It's important to implement security measures. and security measures. This checklist covers essential configurations to ensure your application runs securely in production.

1. **Security headers**
   Security headers are part of the HTTP response sent by your server to a browser. They act as rulesets, dictating:

- Which resources (scripts, images, fonts) can load.
- How browsers should handle HTTPS, cookies, and cross-origin requests.
- What browser features (e.g., camera, geolocation) your site can access.

Setting security headers in your browser handle your site's content and communicate with servers. You can set essential security headers:

```
// next.config.js
/** @type {import('next').NextConfig} */
module.exports = {
  async headers() {
    return [
      {
        source: '/ about',
        headers: [
          {
            key: 'x-custom-header',
            value: 'my custom header value'
          },
          {
            key: 'x-another-custom-header',
            value: 'my other custom header value'
          }
        ]
      }
    ]
  }
}
```

2. Configure CORS

Cross Origin Resource Sharing (CORS) is a security mechanism that's implemented by web browsers that determine which external domains can access your API endpoints. Here's how to configure it in a Next.js application:

```
// pages/api/data.js
export default async function handler(req, res) {
  // Configure CORS headers
  res.setHeader(
    'Access-Control-Allow-Origin',
    'https://trusted-site.com'
  )
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS')
  res.setHeader('Access-Control-Allow-Headers', 'Content-Type')
  // Required if handling authentication
  res.setHeader('Access-Control-Allow-Credentials', 'true')
  // Add security headers
  res.setHeader('X-Content-Type-Options', 'nosniff')
  res.setHeader('X-Frame-Options', 'DENY')
  res.setHeader(
    'Content-Security-Policy',
    "default-src 'self'; script-src 'self' 'unsafe-inline';" +
      "style-src 'self' 'unsafe-inline'; object-src 'none'"
  )
  // Prevents referrer leaks
  res.setHeader('Referrer-Policy', 'no-referrer')
  // Handle preflight (CORS) requests
  if (req.method === 'OPTIONS') {
    res
      .writeHead(204, {
        'Access-Control-Allow-Origin': 'https://trusted-site.com',
        'Access-Control-Allow-Methods': 'GET, POST, OPTIONS',
        'Access-Control-Allow-Headers': 'Content-Type',
        'Access-Control-Allow-Credentials': 'true'
      })
      .end()
    return
  }
  // Validate request method
  const allowedMethods = ['GET', 'POST']
  if (!allowedMethods.includes(req.method)) {
    res.status(405).json({
      error: 'Method Not Allowed'
    })
    return
  }
  try {
    // Your API logic here
    const data = await getData()
    res.status(200).json(data)
  } catch (error) {
    console.error('API Error:', error)
```

```
    res.status(500).json({
      error: 'An error occurred while processing your request'
    })
  }
}
```

3. Implement rate limiting

To implement rate limiting in a Next.js app, you can use a middleware. A proper and scalable implementation should use a shared store in order to properly share the state between instances.

Install the `ioredis` package, then create the following `middleware.js` file in the Next.js root folder.:

```
import { Redis } from "ioredis";

const maxRequestPerIP = parseInt(process.env.MAX_REQUEST_PER_IP || 10);
const rateLimitWindow = parseInt(process.env.RATE_LIMIT_WINDOW || 60);

const redis = new Redis(process.env.REDIS_URL);

export const config = { matcher: "/api/:path*" };

export async function middleware(request) {
  // Get the request IP
  const xff = request.headers.get("x-forwarded-for");
  const ip = xff ? xff.split(",")[0] : "127.0.0.1";

  // Increase the token utilization for the IP
  const key = `ratelimit:${ip}`;
  const used = await redis.incr(key);

  if (used > maxRequestPerIP) {
    return new NextResponse(
      JSON.stringify({ success: false, message: "Too many requests"
      }),
      {
        status: 429,
        headers: {
          "Content-Type": "application/json",
          "X-RateLimit-Remaining": 0,
          "X-RateLimit-Limit": maxRequestPerIP,
        },
      }
    );
  } else {
    // Set the expiration time
```

```
    await redis.expire(key, rateLimitWindow);
  }

  const response = await NextResponse.next();

  // Add rate limit headers to successful responses
  response.headers.set("X-RateLimit-Remaining", maxRequestPerIP - used);
  response.headers.set("X-RateLimit-Limit", maxRequestPerIP);

  return response;
}
```

Next Steps

If your app is deployed to Vercel, Vercel Edge middleware is a great alternative for rate limiting.

For applications using Platformatic Watt and Composer you can implement rate-limiting using the `@fastify/rate-limit` plugin.

Platformatic Composer provides rate limiting through plugins.

5. Create a new Watt application.

```
mkdir new-app
cd new-app
npx wattpm@latest init
```

6. Move your existing application inside the `web` folder.

```
mv ../old-app web/app
```

7. Add a composer using `npx create-platformatic` and choosing the answers like shown below.

```
Need to install the following packages:
create-platformatic@2.45.0
Ok to proceed? (y) y

 Hello Awesome User, welcome to Platformatic 2.45.0
? Where would you like to create your project? .
  Installing @platformatic/runtime@2.45.0...
 Using existing configuration
? Which kind of project do you want to create? @platformatic/composer
  Installing @platformatic/composer@2..0...
? What is the name of the service? composer
? Do you want to create another service? no
? Do you want to use TypeScript? no
[16:06:50] INFO: /home/awesome/new-app/.env written!
[16:06:50] INFO: /home/awesome/new-app/.env.sample written!
[16:06:50] INFO: /home/awesome/new-app/web/composer/package.json
written!
[16:06:50] INFO: /home/awesome/new-app/web/composer/platformatic.json
written!
[16:06:50] INFO: /home/awesome/new-app/web/composer/.gitignore
written!
[16:06:50] INFO: /home/awesome/new-app/web/composer/global.d.ts
written!
[16:06:50] INFO: /home/awesome/new-app/web/composer/README.md
written!
? Do you want to init the git repository? no
  Installing dependencies...
[16:06:52] INFO: Project created successfully, executing post-install
actions...
[16:06:52] INFO: You are all set! Run `npm start` to start your
project.
```

8. Install the required packages:

```
Unset

cd web/composer
npm install @fastify/rate-limit ioredis
```

9. Create the `web/composer/rate-limiter.js` with the following contents:

```
const Redis = require('ioredis')

module.exports = async function (app) {
  const redis = new Redis({ host: 'localhost', port: 6379 })

  return app.register(require('@fastify/rate-limit', {
    max: 100,
    timeWindow: '1 minute'
  })
}
```

10. Modify the `web/composer/platformatic.json` to contain the following settings:

```
{
  "$schema": "https://schemas.platformatic.dev/@platformatic/
  composer/2.45.0.json",
    "composer": {
    "services": [{
      "id": "app",
      "proxy": {
        "prefix": "/app"
      }
    }],
      "refreshTimeout": 1000
  },
  "plugins": {
    "packages": [
      {
        "path": "./rate-limiter.js"
      }
    ]
  },
  "watch": true
}
```

This configuration will limit each IP to 100 requests per minute.

Note: Infrastructure concerns like CDN setup, SSL certificates, and server configuration should be handled separately in your deployment strategy.

Wrapping Up

Building Server-Side Rendered (SSR) frontends with Node.js and frameworks like Next.js offers a compelling blend of performance, SEO benefits, and modern development capabilities.

Throughout this chapter, we explored how SSR enables search engines to efficiently index content while improving initial page load speeds—resulting in a better user experience, particularly for those on slower networks or less powerful devices.

We saw how Node.js, with its event-driven, asynchronous architecture, is a natural fit for SSR, particularly when paired with frameworks like Next.js. Next.js simplifies SSR implementation with automatic routing, data fetching methods like getServerSideProps, and built-in API routes.

It also offers Static Site Generation (SSG) and Incremental Static Regeneration (ISR), allowing developers to fine-tune how pages are rendered and updated.

To put these concepts into practice, we built an event listing application using Next.js 13+ with App Router. This project demonstrated dynamic routing, SSR-powered pages, data fetching strategies, and modern React features like Suspense and streaming. We also covered image optimization with next/image, prefetching for improved navigation, and caching techniques to balance performance with scalability.

Of course, SSR comes with trade-offs. Rendering pages on the server increases CPU and memory usage, making caching strategies, load balancing, and error handling critical for scalability.

We explored different deployment options, including Vercel for ease of use, AWS Amplify for cloud integration, and self-hosting for custom control, as well as Next.js's built-in caching mechanisms for optimizing performance.

By understanding these trade-offs and applying best practices, developers can build fast, SEO-friendly, and scalable web applications that strike the right balance between performance, flexibility, and maintainability.

05

Managing Configurations

Why are configuations important, how to provide them & implement them in Node.js & best practices

05

Managing Configurations

Why are configuations important, how to provide them & implement them in Node.js & best practices

Applications require runtime information to execute specific operations. By leveraging configuration, they can dynamically adapt to different environments and requirements. Common configuration settings include:

Application Behavior Settings
Control how the application runs, including logging verbosity, limits (e.g., connection retries, timeouts), and feature flags.

Application Dependency Settings
Define configurations for external dependencies, such as database connections, third-party URLs, API keys, and other external service settings.

Secrets
Sensitive information related to behavior settings and dependencies, including API keys, authentication tokens, and encryption keys.

A well-structured configuration system centralizes these settings, allowing adjustments without modifying code.

This is crucial for deploying applications across different environments, fine-tuning performance, enhancing resilience, and quickly adapting to evolving business requirements.

Why is Configuration Important?

A robust configuration management strategy ensures your application remains flexible, maintainable, and secure. Achieving these qualities is impossible without a proper configuration implementation.

Flexibility
Configuration allows applications to seamlessly operate across different environments (development, testing, and production) by using environment-specific settings.

Maintainability
Separating configuration from code improves readability, simplifies updates, and streamlines deployments.

Security
Storing sensitive information like API keys and database credentials directly in code poses a security risk. A well-designed configuration system keeps such data external, minimizing the risk of accidental exposure.

How to Provide Configurations

Configurations can be provided in several ways, depending on the application type: typically used.

For Services
Configuration files and environment variables are commonly used.

For Command-Line Applications
Command-line arguments and configuration files are typically used.

Configuration files come in various formats, with .env, JSON, and YAML being among the most popular.

By designing a reliable configuration strategy, you can enhance your application’s adaptability, scalability, and security while simplifying deployment and maintenance.

5.2 The Twelve-Factor Methodology

Building secure, scalable, and maintainable web applications requires following established best practices.

One widely adopted approach is the Twelve-Factor methodology, a set of principles designed to guide modern software development, particularly for Software-as-a-Service (SaaS) applications.

This methodology addresses both backend development and development operations (DevOps), ensuring efficiency and scalability.

Adopting the Twelve-Factor methodology offers several benefits:

Easy Onboarding
Declarative formats enable automated setup, making it easier for new developers to get started.

Portability
The application maintains a seamless relationship with the underlying operating system, ensuring predictable deployment across different environments.

Cloud Readiness
Applications built with Twelve-Factor principles are designed for smooth deployment in cloud environments.

Scalability
The methodology encourages stateless processes and externalized configurations, ensuring efficient scalability.

One of the core principles of the Twelve-Factor methodology is configuration management. It emphasizes that configurations should be stored in the environment, separate from the codebase.

What is an Environment in Software Development?

In software development, an “environment” refers to the setup or context in which software is developed, tested, and deployed. It encompasses the hardware, software, integrations, and configurations that allow an application to run and operate.

Different environments are used at various stages of the software development lifecycle to ensure smooth transitions from writing code to running it in production.

Each environment is designed to fulfill a specific role, offering different resources and settings tailored to the needs of that stage. They enable developers, testers, and operations teams to build, refine, to finally deploy the software for the users with minimal risk.

Understanding and properly managing these environments is essential for building reliable, scalable applications while maintaining consistency between development and production.

The most common types of environments include:

Local
The developer’s personal machine, where individual code is written, debugged, and tested in isolation. This environment is often used for early-stage development and quick iterations before sharing code with others.

Development
A shared environment where developers integrate their work into a common codebase. It allows for collaboration on features, with developers often using it to test new features before moving to testing or staging. It might be configured to simulate certain production-like conditions but is usually less restrictive.

Testing
A controlled space used for rigorous testing. Automated and manual tests are run here to verify the functionality, stability, and performance of the software. It may involve unit tests, integration tests, and other validation mechanisms to catch bugs early and ensure quality.

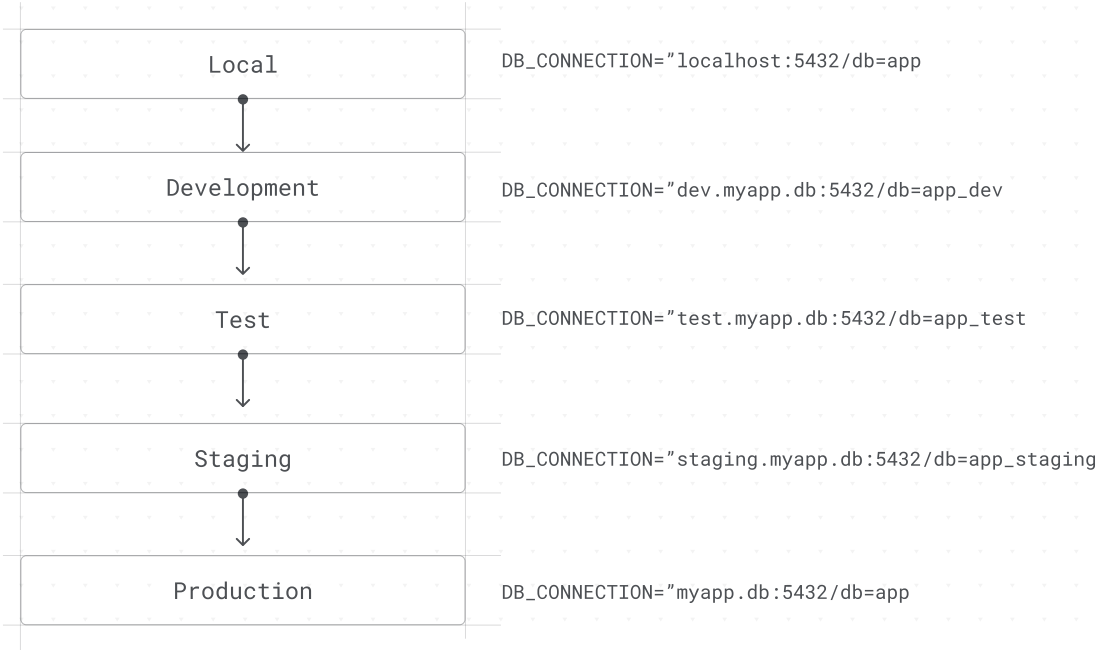
Staging

A pre-production environment that closely mirrors production. It is used for final testing and user acceptance testing (UAT) to ensure that the application performs as expected under conditions similar to the live environment. Staging serves as a safety net before deploying to production.

Production

The live environment where end users interact with the application. This is the most critical environment, and it must be highly reliable, secure, and optimized for performance. Any issues here can directly impact users, so it should always be carefully monitored and maintained.

To understand the importance of configuration in the development lifecycle, let's consider a typical backend application with its own database. Each environment—local, development, testing, staging, production—requires a replica of the database with different settings and access configurations. For example, the database pseudo connection string for each environment might be as follows



The application in each environment needs to adapt to the appropriate connection string, ensuring it connects to the correct database. This is a classic example of how configuration management is crucial for the smooth functioning of an application across different environments.

Note also CI/CD systems often require specific configuration values or flags to set up the context for testing and building the application.

CI pipelines typically operate in a controlled environment, where configurations need to be explicitly defined for each stage of the pipeline (e.g., build, test, deploy).

This ensures that the application behaves correctly within the testing and deployment contexts, with the appropriate resources and services accessible during each phase.

5.3 How to provide Runtime Configurations in a Node.js Application

Now that we've covered the fundamental concepts of configuration management, let's explore how to provide runtime configuration settings in a Node.js application.
Node.js offers built-in mechanisms for handling runtime configurations through:

Adopting the Twelve-Factor methodology offers several benefits:

- `process.env` – Retrieves environment variables.
- `process.argv` – Captures command-line arguments.

Using Environment Variables (process.env)

The process.env object contains the current environment variables as key-value pairs. These variables can be used to configure application behavior dynamically without modifying the code.

Here's an example of what process.env might look like:

```
{
  "USER": "alice",
  "PATH": "~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin",
  "PWD": "/home/alice",
  "HOME": "/home/alice"
}
```

Using Command-Line Arguments (process.argv)

The process.argv property is an array containing command-line arguments that are passed to the Node.js process when it starts.

- The first element (`process.argv[0]`) is the path to the Node.js binary.
- The second element (`process.argv[1]`) is the path to the JavaScript file being executed.

Any additional elements are user-defined command-line arguments.

For example, consider the following command:

```
LOG_LEVEL=debug node download.mjs http://someurl.com/file
```

In this case:

- `process.env.LOG_LEVEL` is set to “debug”, which can control logging behavior.

process.argv will contain:

```
[
  '/usr/local/bin/node',  // Node.js executable path
  '/path/to/download.mjs', // Path to the script
  'http://someurl.com/file' // User-provided argument
]
```

Example: Handling Runtime Configuration in a Script

Here’s how we can use process.env and process.argv to configure a simple ESM script dynamically:

For example, consider the following command:

```
const url = process.argv[2]; // The URL is the third argument
if (!url) {
  console.error("Error: No URL provided.");
  process.exit(1);
}

if (process.env.LOG_LEVEL === "debug") {
  console.log("Starting download: " + url);
}
try {
  const response = await fetch(url);
  const data = await response.text();
  console.log("Download complete:", data);
} catch (error) {
  console.error("Download failed:", error));
}
```

To handle configuration more systematically through command-line arguments, we can use `parseArgs` from `node:util` or leverage a library such as yargs or commander.

These libraries allow us to define structured command-line options and access arguments more conveniently. For example, we can create a CLI tool to calculate the fibonacci sequence

```
#!/usr/bin/env node
import { parseArgs } from 'node:util'
try {
  const options = {
    number: { type: 'string' },
    help: { type: 'boolean', short: 'h' }
  }
  const { values } = parseArgs({ options });
  if (values.help) {
    console.log(`
Usage: fibo --number <number>


Options:
 --number       Number to calculate the Fibonacci sequence
   `)
    process.exit(0)
```

```
  }


  const number = parseInt(values.number)
  if (isNaN(number) || number < 0) {
    console.error('Error: Number is invalid')
    process.exit(1)
  }


  const fibo = (value) => {
    if (value <= 1) return value
    return fibo(value - 1) + fibo(value - 2)
  }


  console.log(`The Fibonacci sequence is: ${fibo(number)}`)
} catch (error) {
  console.error('Error parsing arguments:', error.message)
  process.exit(1)
}
```

And run it as

```
./fibo --number=1
```

5.4 Configuration Files

When an application requires multiple configuration values, passing them all via environment variables or command-line arguments becomes impractical. A more structured approach would be to use configuration files, allowing settings to be stored in a single file that the application loads at startup.

The most common practice is to use a .env file, which can be loaded with the dotenv https://www.npmjs.com/package/dotenv library. However, starting from Node.js v20, `.env` files can be natively loaded using the `--env` option, eliminating the need for an external library.

Regardless of the approach, the .env file is read at startup, and its contents are injected into process.env, potentially overriding any pre-existing environment variables.

.env File Format

A `.env` file follows a simple key-value format, similar to shell environment files:

```
DB_HOST=localhost
DB_PORT=5432
DB_USER=myuser
DB_PASSWORD=supersecret
LOG_LEVEL=debug
```

Best Practices for `.env` Files

Never commit `.env` files to version control, as they often contain sensitive information that could be accidentally exposed, **posing a severe security risk**. Instead, always add them to `.gitignore`.

Provide a `.env.sample` file with all required environment variables and placeholder values. This improves the developer experience (DX) by helping new developers quickly set up their environment.

Avoid using and committing `.env` files per environment (e.g., `.env.dev`, `.env.test`). Configuration management should be decoupled from the code, with environment variables provided externally.

Moreover, configuration must not include any application data that directly or indirectly depends on the environment. Such data belongs to the application's storage layer and must be handled separately.

Mixing environment-dependent data with configuration can lead to confusion, unnecessary complexity, and potential misconfigurations.

Other Configuration File Formats

Besides `.env` files, applications may use other formats for configuration, the most popular are JSON or YAML.

While this is a valid design choice, mixing multiple sources adds complexity. The recommended approach is to keep configuration management simple and consistent by using a single, well-defined configuration format.

5.5 How to Properly Implement Configuration in Node.js

The most basic way to provide settings to an application is by hardcoding them directly into the code. However, this is a common 'code smell' that should be avoided in professional software development.

Hardcoding creates security risks, as sensitive information like secrets are tightly coupled with the code and exposed to anyone with access. It also decreases codebase maintainability, as settings are often repeated across the codebase, making them difficult to change and prone to inconsistencies. This approach can be seen as a failure to implement proper configuration management.

If we go back to the Twelve-Factor methodology, one of its core principles is configuration management. The methodology emphasizes that configurations should be stored in the environment, rather than being hardcoded or directly included in the codebase.

In this section, we'll explore common anti-patterns in configuration management within Node.js and review best practices for managing configurations securely and efficiently.

Anti-pattern: Global Configuration

While it's common for an application to have a global configuration, adopting a global configuration scope loaded from every script can be considered an anti-pattern because global variables are viewed as poor design practices.

Although using a single module to hold configuration settings might seem convenient, it can hinder the creation of modular and composable code.

Note: The recommended design approach is to explicitly pass configuration values as inputs to the modules, classes, or functions that require them. This decouples the application's overall configuration from individual module options, allowing greater flexibility.

Any code entry point should have its own options, ensuring that each part of the application operates independently without relying on a shared global state.

Additionally, this approach makes it easier to inject different configurations for testing purposes, improving maintainability and reducing unintended dependencies.

Using a global configuration introduces several challenges that negatively impact modularity, maintainability, and testability.

One of the main issues is limited modularity and composability. When configuration is stored in a global object or module, components become tightly coupled to it, making it harder to design reusable, self-contained, and easily composable modules.

Another significant drawback is the difficulty in testing and maintenance. Since configuration is shared globally, different test cases may unintentionally interfere with one another by modifying the global state. Writing isolated unit tests becomes more complex, as each test might need to reset or mock the global configuration. Additionally, changes to configuration logic can introduce unexpected side effects throughout the application.

Hidden dependencies are another problem. When modules implicitly rely on a global configuration object, it obscures dependencies, making the code harder to understand, refactor, and maintain. Without explicitly passing configuration where needed, it becomes unclear which parts of the application depend on specific settings.

A global configuration also reduces flexibility. It becomes challenging to override settings for specific parts of the application, which is particularly problematic when running multiple instances of a service with different configurations. This often forces developers to implement workarounds that increase complexity.

In dynamic environments, there is also a risk of race conditions. If configuration values are modified at runtime—for example, when settings are reloaded dynamically—different parts of the application may read inconsistent data, leading to unpredictable behavior and hard-to-debug issues.

Lastly, security vulnerabilities arise when sensitive information, such as API keys and credentials, is stored in a global object.

Any part of the application can access it, increasing the risk of accidental leaks or unauthorized access.

Let's take a look at an example.

In lib/config.js we define the config values:

```
export const config = {
  database: {
    host: process.env.DB_HOST || 'localhost',
    port: process.env.DB_PORT || 5432,
    user: process.env.DB_USER || 'postgres',
    password: process.env.DB_PASSWORD || 'password',
    name: process.env.DB_NAME || 'myapp'
  }
 };
```

In lib/database.js we define the database handler, which directly imports the global config. As we can see, this way creates tight coupling from the "connect" function and the config

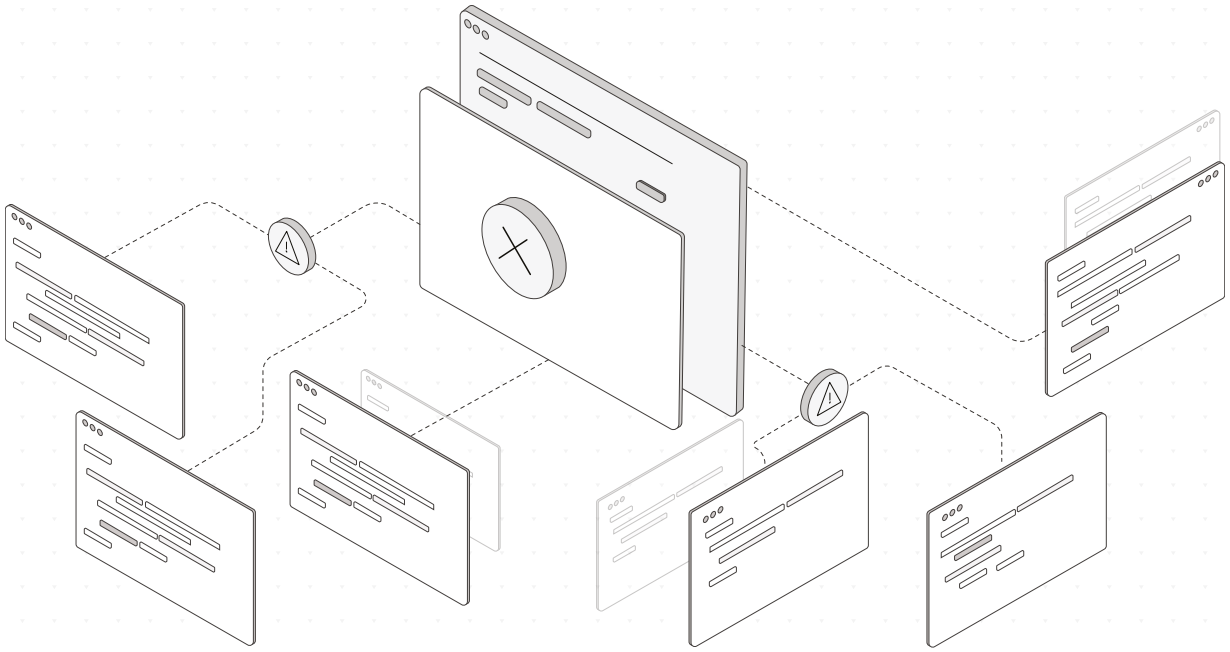
```
import config from './config';


function connect() {
  const { host, port, user, password, name } = config.database;
  const pool = new Pool({
    host,
    port,
    user,
    password,
    database: name
  });
  return pool;
}
```

Now, let's examine the proper implementation to address this design issue. In `lib/database.js`, rather than importing a global configuration file, the function accepts configuration values as parameters, conventionally called `options` ensuring flexibility while keeping the configuration structure unchanged.

```
export function connect(options) {
  const { host, port, user, password, name } = options;
  console.log(`Connecting to database ${name} at
  ${host}:${port} with user ${user}`);
  ...
};
 }
```

So, in the caller, for example, `app.js`, we call it as follows:

```
import { config } from './lib/config';
import database from './lib/database';


function start() {
  const db = database.connect(config.database);
    ...
}
```



5.6 Anti-pattern: process.env.NODE_ENV Static Pollution

NODE_ENV is an environment variable in Node.js that specifies whether the application is running in a production environment.

This pattern uses the **NODE_ENV** variable to identify the current working environment, typically set to `development`, `test` or `production`. It directly depends on this value to determine the application's behavior, such as optimizing performance, enabling or disabling debugging tools, or activating environment-specific features. The biggest problem of this practice is that development in this context signifies the `"local"` environment from an infrastructure point of view. As one can expect, when two teams disagree on terminology, significant problems arises.

The adoption of **NODE_ENV** to define the application's runtime environment became a de facto standard in the Node.js ecosystem, largely due to its early use in the connect library and later in the Express framework, one of the first widely adopted web frameworks for Node.js, gained significant traction as Node.js grew in popularity after 2010. Later, it was used by React to enable/disable some optimizations and debugging tools.

This convention was inspired by earlier software engineering practices, such as the use of environment variables in Unix systems and frameworks like Ruby on Rails, which employed similar variables (e.g., **RAILS_ENV**) to manage runtime environments. Over time, many libraries and frameworks—such as Nest.js and tRPC —have adopted this convention, prompting developers to implement conditional logic based on the environment.

However, when such code snippets are scattered across the codebase, maintaining the application becomes increasingly challenging.

```
if (process.env.NODE_ENV === “production”) {
  enableOptimizations();
} else {
  enableDebuggingTools();
}
---
if (process.env.NODE_ENV === “production”) {
  enableRateLimiting();
} else {
  disableRateLimiting();
}
---
if (process.env.NODE_ENV === “test”) {
  useMockData();
} else {
  useRealData();
}
```

Relying on **NODE_ENV** to manage configurations throughout an application is an anti-pattern that can introduce several significant issues.

One major issue is the scattering of configuration logic across the codebase. Instead of centralizing configuration settings, this approach embeds environment-specific logic within multiple modules, making the application more difficult to maintain, refactor, and debug. Furthermore, since different environments may have varying configurations, inconsistencies can arise, increasing the likelihood of errors.

Another issue is the tight coupling of the code to the execution environment. Frequent checks against `process.env.NODE_ENV` violates the Separation of Concerns[1] principle– which outlines that applications should not be written as one solid block– by intertwining business logic with environment-specific settings. A more effective approach is to inject the appropriate configuration at startup rather than querying environment variables dynamically during execution.

This approach also complicates testing and debugging. When application behavior is dictated by **NODE_ENV**, developers must manually set environment variables to simulate different conditions. Additionally, misconfigurations can persist across processes, leading to unintended side effects in unrelated tests. Debugging failures can also become non-deterministic due to inconsistencies between different environments.

Moreover, managing configurations through **NODE_ENV** introduces limited flexibility. Modern applications typically require more than just “development” and “production” environments. Staging, QA, feature preview, and customer-specific environments are

[1] *https://nalexn.github.io/separation-of-concerns/*

common in many projects. Hardcoding logic based on a single environment variable does not scale as configuration complexity grows.

Finally, there are security risks associated with relying on **NODE_ENV** for security-sensitive behavior. It is a common but incorrect assumption that setting `NODE_ENV="production"` automatically enforces a secure configuration. Instead, applications should explicitly load secure defaults from an external configuration source rather than depending on a single environment variable.

Despite its limitations, `NODE_ENV=production` remains a historical convention and is often required in production configurations for performance optimizations. However, it should only be used in actual production deployments, ensuring that critical environment-specific settings are managed properly.

A common mistake is setting `NODE_ENV=staging` or `NODE_ENV=production` in non-production environments, assuming it will replicate production behavior. This can lead to unintended consequences, such as:

- Missing detailed error messages, making debugging more difficult.
- Skipping development-only dependencies, potentially causing broken builds or missing functionality in local or test environments.

To effectively address this issue and provide the flexibility to enable or disable features, feature flags offer an ideal solution. Instead of relying on a single environment variable, we define fine-grained feature flags, each tied to its own environment variable. This allows explicit control over application behavior based on these values.

For example:

```
if (config.enableDebuggingTools) {
  enableDebuggingTools();
}

if (config.enableRateLimit) {
  enableRateLimiting();
} else {
  disableRateLimiting();
}

if (config.useMockData) {
  useMockData();
} else {
  useRealData();
}
```

We can define the respective environment variables for different environments such as **development**, **test**, or **production**. This approach ensures flexibility and allows clear control over features without requiring code changes, even when we need to enable rate limiting in development to debug it, for instance. It also enables easy introduction of new environments by simply adjusting the configuration.

Here's how you can set this up in your configuration file, with **production** serving as the default configuration:

lib/config.js

```
export const config = {
  useMockData: process.env.USE_MOCK_DATA === 'true',
  enableDebuggingTools: process.env.ENABLE_DEBUGGING_TOOLS === 'true',
  enableRateLimit: process.env.ENABLE_RATE_LIMIT === 'true',
};
```

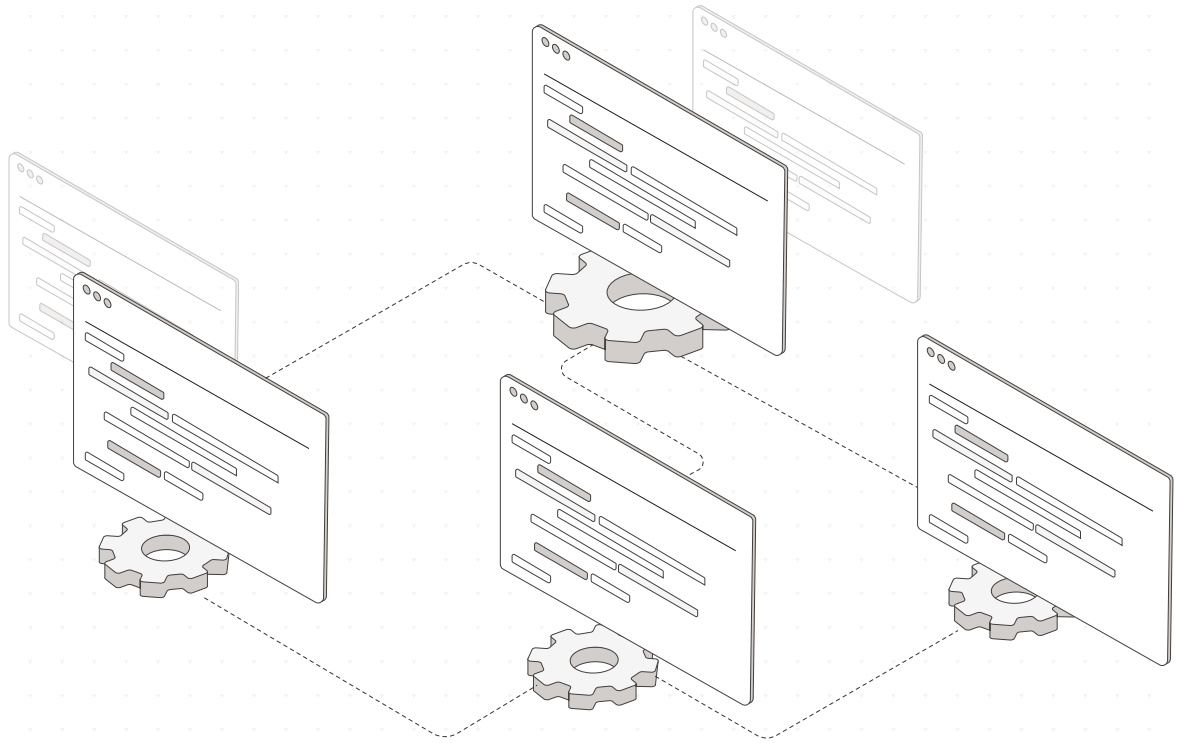
Configuration for Development:

```
USE_MOCK_DATA=true
ENABLE_DEBUGGING_TOOLS=true
ENABLE_RATE_LIMIT=false
```

Configuration for Production:

```
USE_MOCK_DATA=false
ENABLE_DEBUGGING_TOOLS=false
ENABLE_RATE_LIMIT=true
```

This ensures that the application behaves differently across environments without requiring changes to the codebase.



5.7 Anti-pattern: Hierarchical Configurations

A historical anti-pattern in Node.js, popularized by the config library, is the use of hierarchical configurations. This method involves committing multiple configuration files to the repository and selecting the appropriate one based on the **NODE_ENV** variable. While this approach helps avoid polluting the code with **NODE_ENV** checks, it introduces several new problems.

One such problem is the encouragement of committing configuration files to the repository, which can lead to sensitive or environment-specific settings being accidentally included. This inadvertently increases the risk of exposing secrets. To mitigate this risk, some developers resort to encrypting and committing production-specific configuration files. However, this creates a poor developer experience (DX) and raises the question: why only encrypt production files? Other configurations might contain sensitive data as well, making this approach incomplete and insufficient for securing all types of configuration data.

Instead of defining a default configuration and overriding values as needed, hierarchical configurations enforce a rigid structure that is harder to manage, adding unnecessary complexity. In reality, a single configuration file with sensible defaults, combined with external overrides (e.g., environment variables, secrets management tools), is a more straightforward approach.

Configuration consistency is not automatically enforced, as different environment-specific files can lead to discrepancies between settings. Without clear enforcement, these different configurations may diverge from one another, resulting in unpredictable behavior across environments.

Moreover, the config module operates globally within the application. This makes it difficult to manage configurations in a modular or isolated manner, increasing the complexity and making the configuration more prone to errors as the application grows.

Example:
A common setup for configuration involves the following steps:

1. The config library initially loads the configuration from ./config/default.json.
2. It then loads the environment-specific file ./config/{NODE_ENV}.json, overriding values from the default configuration.
3. If the NODE_ENV is not set, it defaults to loading the development configuration.

```
- config /
  - default.json
    - development.json
    - production.json
```

---

config/default.json - Default Configuration

This file contains the common configuration shared across all environments:

```
{
  “server”: {
    “port”: 3000,
    “host”: “localhost”
  },
  “database”: {
    “connectionString”: “localhost”,
    “port”: 5432,
    “name”: “myapp”,
    “user”: “dbuser”,
    “password”: “default_password”
  },
  “logging”: {
    “level”: “info”
  }
}
```

config/development.json - Development Configuration

A configuration specifically for the development environment, overriding the default ones, but also adding a new values under a new “features” property

```
{
  “server”: {
    “port”: 3001
  },
  “database”: {
    “password”: “dev_password_123”
  },
  “logging”: {
    “level”: “debug”
  },
  “features”: {
    “enableDebug”: true,
    “mockServices”: true
  }
}
```

config/production.json - Production Configuration

A configuration specifically for the production environment. It follows the development “features” properties, that may not be present in “default”, adding another property “cache” that is not present neither in “default” or “development”.

Not the primary security issue: passwords are stored in plain text, making them accessible to anyone with access to the code.

This poses a serious security risk and also constitutes a legal violation, such as non-compliance with GDPR.

```
{
  “server”: {
    “port”: 80,
    “host”: “0.0.0.0”
  },
  “database”: {
    “host”: “prod-db.example.com”,
    “password”: “prod_super_secret_password_xyz” // plain password!
  },
  “logging”: {
    “level”: “warn”
  },
  “features”: {
    “enableDebug”: false,
    “mockServices”: false
  },
  “cache”: {
    “redis”: {
      “host”: “redis.prod.example.com”,
      “password”: “redis_prod_password” // plain password!
    }
  }
}
```

app.js - Application file

Within the application, developers must be cautious when accessing configuration values through strings, as this can lead to typographical errors and runtime issues due to the absence of guarantees about correctness.

Additionally, checking the presence of configuration sections adds unnecessary verbosity to the code, making maintenance more difficult.

The global accessibility of configuration reduces modularity, complicating the isolation and management of different parts of the application.

Furthermore, this approach presents a security risk, as it can inadvertently expose sensitive information. For instance, calling config.get(‘cache.redis’) unintentionally return and display the password it contains.

```
'use strict'
const config = require('config')
const fastify = require('fastify')({ logger: true })
// Start the server
const start = async () => {
  // Log the current environment
  console.log(`Running in ${process.env.NODE_ENV || 'default'}
  environment`)
  // The config library automatically loads configuration files based
  on NODE_ENV:
  // 1. It first loads ./config/default.json
  // 2. Then it loads ./config/{NODE_ENV}.json and overrides values
  from default
  // 3. If NODE_ENV is not set, it defaults to 'development'
  console.dir({
    environment: process.env.NODE_ENV || 'default',
    config: {
      server: {
        host: config.get('server.host'),
        port: config.get('server.port')
      },
      database: {
        port: config.get('database.port'),
        name: config.get('database.name'),
        user: config.get('database.user'),
        password: '***'
      },
      logging: {
        level: config.get('logging.level')
      },
      // Features may only exist in certain environment configs
      features: config.has('features') ? {
        enableDebug: config.get('features.enableDebug'),
        mockServices: config.get('features.mockServices')
      } : 'Not configured for this environment',
      // Cache may only exist in production config
      cache: config.has('cache') ? {
        redis: config.get('cache.redis')
      } : 'Not configured for this environment'
    }
  }, { depth: null })
  try {
    const port = config.get('server.port')
    const host = config.get('server.host')


    await fastify.listen({ port, host })
  } catch (err) {
    fastify.log.error(err)
    process.exit(1)
  }
}
start()
```

This pattern hinders team efficiency and compromises your application's security. It must be avoided, as it slows your team down and increases security risk.

Later in this chapter, we'll explore a cleaner approach to handling configurations without relying on this pattern.

Best Practices for Configuration Management

Now that we've explored common anti-patterns, let's focus on best practices for managing configurations in a Node.js application.

The goal of these best practices is to:

Allow configurations to grow and adapt as the application evolves.

Facilitate a seamless transition across different environments, from development to production.

Detect issues early, minimizing the risk of misconfigurations.

Enable quick and safe configuration changes without causing disruptions or unnecessary complexity.

By following these principles, we can build a scalable, maintainable, and secure configuration system, because excellent DX starts with proper configuration management.

Let's explore the recommended approach, and then see how to apply all of these best practices in a practical example.

Single Source of Truth

The most effective way to manage configuration is to centralize it in a single file, thereby avoiding scattered configuration logic throughout the codebase and ensuring that all configuration values are gathered in one location.

This configuration file should handle all operations related to configuration, including the collection of environment variables, loading from files, defining optional settings, setting default values, and performing validation. By consolidating all configuration-related tasks into one file, developers can be confident that there are no unmanaged or inconsistent configuration elements within the codebase.

A common approach is to create a config.js file in the application code, where all configuration values are defined and exported.

This ensures:

Consistency
All settings are in one place, making it easy to manage and update.

Maintainability
Developers don't have to search across multiple files to modify configurations.

Separation of Concerns
The application logic remains independent of environment-specific settings.

This approach creates a clear and structured configuration system, improving DX and reducing the likelihood of misconfigurations.

Default Values

Setting default values for configuration is a best practice, especially for Application Behavior Settings. These include parameters such as connection timeouts, retry limits, and logging levels, which can be predefined in the configuration file and overridden when necessary.

Default values should always be aligned with the production environment, as it is the primary deployment target. Development and staging environments should explicitly override these defaults as needed.

While application behavior settings can have defaults, dependency settings—such as database credentials, API keys, and secrets—should never have default values.

Providing defaults for these critical settings could lead to unintended behavior, such as accidentally using production settings in development environments.

Instead, the application should fail to start if these vital settings are missing, throwing an error to ensure the issue is addressed before deployment.

Validation

After inputting configuration values, we must ensure they are correct at runtime. A good way to do so is by validating environment variables.

When the validation fails, it will throw an error with details of the incorrect values

Various libraries can help handle this, including env-schema, zod, and TypeBox.

Note: handling non-string values is a common issue when working with environment variables. Since all environment variables are inherently treated as strings, values like booleans need to be explicitly converted.

For example, “true” or “false” as strings must be parsed into their respective boolean types (true/false). Proper conversion ensures that the application behaves as expected when using non-string values like numbers, booleans, or arrays.

Using `env-schema`

`env-schema` is a lightweight library that validates environment variables based on a JSON schema. It ensures that required values are present and correctly formatted.

Example: lib/config.js

```
import envSchema from 'env-schema'


const schema = {
  type: 'object',
  properties: {
    PORT: {
      type: 'number',
      minimum: 1, maximum: 65535,
      default: 3000,
    },
    HOST: {
      type: 'string',
      default: '127.0.0.1',
    },
    LOG_LEVEL: {
      type: 'string',
      enum: ['fatal', 'error', 'warn', 'info', 'debug', 'trace'],
      default: 'info',
    },
    API_KEY: {
      type: 'string',
      minLength: 10,
    },
  },
  required: ['API_KEY']
};


const config = envSchema({
  schema: schema,
  dotenv: true
});
```

Note: `dotenv: true` automatically loads values from a `.env` file.

Combining `env-schema` with TypeBox for Type Safety

Since `env-schema` does not provide TypeScript types, it can be combined with TypeBox for type safety. This ensures that TypeScript enforces the structure at compile time.

Example: lib/config.ts

```
import envSchema from 'env-schema'
import { Type, type Static } from '@sinclair/typebox'


const schema = Type.Object({
  PORT: Type.Number({
    minimum: 1,
    maximum: 65535,
    default: 3000,
  }),
  HOST: Type.String({
    default: '127.0.0.1',
  }),
  LOG_LEVEL: Type.Enum({
    fatal: 'fatal',
    error: 'error',
    warn: 'warn',
    info: 'info',
    debug: 'debug',
    trace: 'trace',
  }, {
    default: 'info',
  }),
  API_KEY: Type.String({
    minLength: 10,
  }),
}, {
  required: ['API_KEY']
});


type Schema = Static<typeof schema>


const config = envSchema<Schema>({
  schema,
  dotenv: true
});
```

Using zod

A more type-friendly alternative is zod, which provides a fluent API for defining and validating schemas.

```
import { z } from 'zod'
import dotenv from 'dotenv'


dotenv.config()


const schema = z.object({
  PORT: z.coerce
    .number()
    .int()
    .min(1)
    .max(65535)
    .default(3000),
  HOST: z.string().default('127.0.0.1'),
  LOG_LEVEL: z.enum(['fatal', 'error', 'warn', 'info', 'debug',
  'trace'])
    .default('info'),
  API_KEY: z.string().min(10),
});
const config = schema.parse(process.env);
```

Runtime Validation

While static validation ensures that environment variables are correctly formatted and present, it does not guarantee that the values are valid in a real-world context. Runtime validation overcomes this limitation by ensuring that environment variables, although syntactically correct, also hold practical validity during application execution.

For instance, a database connection string or an external URL may pass static syntax checks but could still fail at runtime if the database is unreachable or if the URL is unavailable.

To manage this effectively, it is advisable to delegate the responsibility of validating certain configuration values to the modules that will use them at runtime. These modules often possess the necessary context, including client instances and custom logic—such as retry mechanisms—to perform the validation accurately. Including these

checks directly in the configuration layer can introduce unnecessary complexity due to external dependencies.

By combining static validation with runtime checks, we ensure that configuration values are not only correctly formatted but also practically valid, reducing the risk of issues arising during the application's execution.

Segregation

Segregation in configuration management involves logically splitting the configuration object and passing only the relevant parts to the modules, objects, or functions that need them. Typically, the settings passed to modules are referred to as options, while the broader settings that govern the main application are referred to as configuration.

This approach ensures that different parts of the application have access only to the configuration values they need, thus enhancing security. This also minimizes the amount of data passed to each module, ensuring that only the smallest, most relevant subset of configuration is provided.

By following this principle, we avoid unintended exposure of sensitive or unnecessary configuration values to modules that don't require them, promoting a more secure and maintainable application design.

Sometimes, the application design requires accessing multiple parts of the configuration. In such cases, we can:

- Merge multiple subparts into a single option
- Transform the configuration to map to the module options

We should avoid passing the entire configuration to modules, even if this approach introduces some logic and overhead. This ensures that the configuration is used securely and efficiently, without unnecessary complexity.

Readonly

Making the configuration read-only is an additional safety measure to prevent unintended modifications during runtime. In Node.js, the `Object.freeze()` method can be used to deep-lock the configuration object, ensuring that its values cannot be altered.

Once the configuration is frozen, any attempt to modify its properties will result in a runtime error, effectively safeguarding the configuration from accidental changes. This practice also eliminates the need to create deep copies of the configuration, which can be computationally expensive and inefficient during execution.

By enforcing immutability on the configuration object, we enhance the application's predictability and security, ensuring that configuration values remain consistent and unaltered throughout the program's runtime.

Handle secrets

Handling secrets requires careful attention to ensure they remain secure and are not inadvertently exposed. The common practice of passing secrets via environment variables at runtime is inherently risky since environment variables can be accessed by anyone with the ability to inspect the running process.

If an attacker gains access to a pod or server, they can retrieve these values using commands like `ps` or by reading from `/proc/<pid>/environ`. Additionally, environment variables are often visible in container configurations and pod specifications. For example, running `kubectl describe pod` can reveal environment variables in plaintext.

To mitigate these risks, tools like AWS Secrets Manager, HashiCorp Vault, and Kubernetes Secrets allow secrets to be securely stored and injected into applications at runtime without exposing them through environment variables.

Among these, HashiCorp Vault provides a comprehensive solution for managing sensitive data such as API keys, database credentials, certificates, and encryption keys. Operating under the "zero trust" principle, Vault ensures that no system, user, or application is implicitly trusted; access is granted strictly based on authentication and authorization policies.

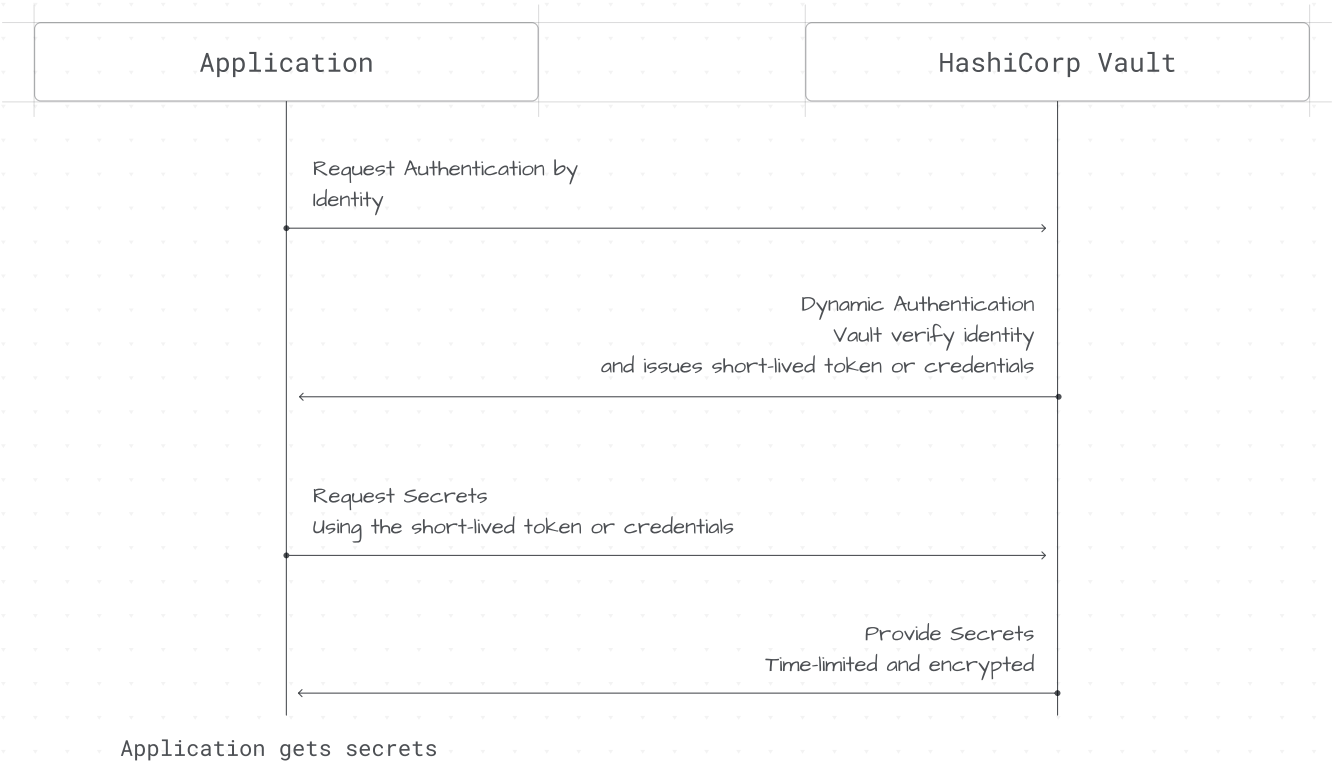
The "Secret Zero" Problem

One of the primary challenges in secret management is the "secret zero" problem, which refers to the dilemma of how to securely authenticate and retrieve secrets initially, without relying on hardcoded credentials. The core issue lies in the fact that to retrieve or access other secrets, you first need a secret to get started. If that initial secret (the "secret zero") is stored insecurely or hardcoded into your application, it defeats the purpose of securing the rest of your secrets.

Vault tackles this challenge by offering dynamic authentication mechanisms. Instead of relying on static, hardcoded credentials (which are often exposed or compromised), Vault provides a system where credentials are dynamically generated, typically based on the identity of the requester. This means that each time the application or service needs to authenticate, it can request temporary, one-time-use credentials or tokens from Vault. These credentials are time-limited and do not require long-term storage, thus removing the risk of exposing secrets in your codebase.

To maintain security, Vault implements mechanisms such as leasing and automatic revocation. Every secret issued by Vault has an expiration period, requiring clients to renew access periodically. If a secret is compromised or no longer needed, Vault can revoke it immediately to prevent unauthorized access. The system also enforces a sealed and unsealed state, using Shamir's Secret Sharing[2] to protect the master key. Upon startup, Vault remains sealed and must be unsealed with a threshold number of key shares before it can decrypt and serve secrets.

By enforcing strict access controls, issuing short-lived credentials, and providing encryption, HashiCorp Vault offers a secure, centralized solution for managing secrets while eliminating the risks associated with hardcoded credentials and uncontrolled secret proliferation.

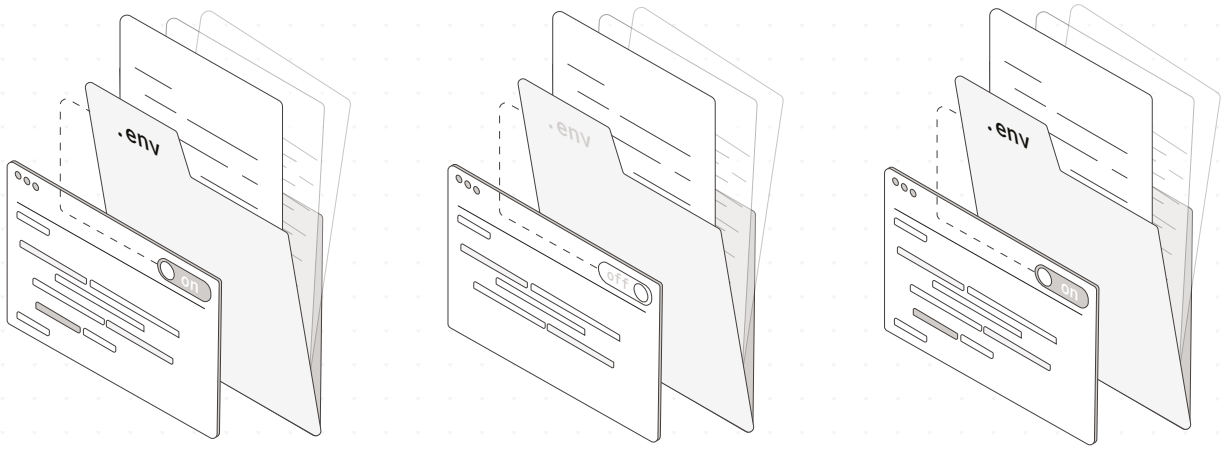


[2] https://www.geeksforgeeks.org/shamirs-secret-sharing-algorithm-cryptography/

Once secret access is managed at the infrastructure level, it becomes the developer's responsibility to ensure that secrets remain secure within the application code.

Developers must take care to prevent secrets from being inadvertently exposed through logs. If secrets must be logged for debugging purposes, they should always be redacted or masked before being output.

To maintain flexibility during development, secrets can still be retrieved from environment variables when necessary, with access controlled through feature flags. In the final example of this chapter, we will demonstrate how to properly configure an application to handle secrets securely and efficiently.



5.8 Environment Variables as Features flags

Feature flags, also known as feature toggles, provide an elegant and effective way to manage different behaviors of an application at runtime.

One of the main advantages of feature flags is that they eliminate the need to rely on environment variables like **NODE_ENV** to control application behavior. By decoupling configuration from the code logic, feature flags centralize feature management. This not only cleans up the codebase but also makes it more maintainable.

Additionally, feature flags provide explicit control over which features are enabled or disabled, reducing the risk of unintended behaviors or hidden changes in the application. This approach ensures that modifications to features are intentional, and makes it easier to manage complex configurations, as each feature can be toggled independently, without requiring code changes.

Moreover, feature flags provide a clear mechanism for rolling out new features incrementally, testing new functionalities in production, or quickly rolling back changes if something goes wrong, all without deploying new versions of the code.

Implementing Configuration Best Practices

Now that we've explored the recommended best practices for managing configuration, let's discuss how to implement them in a practical setting. The following steps will guide you through establishing a well-structured, scalable, and secure configuration management system in your application.

The application structure will look like this:

```
.
├── lib
│   ├── app.js
│   ├── config.js
│   └── logger.js
├── plugins
│   └── db.js
└── main.js
```

Configuration (lib/config.js)

The configuration module implements several best practices:

- **Schema Validation:** Uses `zod` for runtime type checking and validation
- **Environment Variables:** Configuration is sourced from environment variables
- **Type Coercion:** Automatic type conversion for numeric values
- **Default Values:** Sensible defaults for optional configurations
- **Secrets Management:** Optional HashiCorp Vault support for secrets management
- **Feature Flags:** Structured approach to feature management
- **Immutability:** Configuration object is frozen using `Object.freeze`

Key implementation:

```
import { z } from 'zod';

const configSchema = z.object({
  app: z.object({
    port: z.coerce.number().min(1024).max(65535).default(3000),
    host: z.string().default('localhost'),
  }),
  log: z.object({
    level: z.enum(['fatal', 'error', 'warn', 'info', 'debug',
    'trace']).default('info'),
    pretty: z.enum(['', 'true', 'false']).optional()
      .transform((val) => val === 'true'),
    redactions: z.array(z.string()).default([]),
  }),
  db: z.object({ connectionString: z.string().min(1) }),
  features: z.object({
    exposeConfig: z.enum(['', 'true', 'false'])
      .optional().transform((val) => val === 'true'),
    allowedOrigins: z.string().default('localhost:3000').
    transform(str => str.split(',')),
    useVault: z.enum(['', 'true', 'false'])
      .optional().transform((val) => val === 'true')
  })
});

// Vault configuration schema
const vaultConfigSchema = z.object({
  addr: z.string().url().default('http://127.0.0.1:8200'),
  role: z.string().default('myapp'),
  tokenPath: z.string().default('/var/run/secrets/kubernetes.io/
  serviceaccount/token'),
  secretPath: z.string().default('secret/data/myapp')
});

export async function getConfig() {
  // Fetch secrets from vault
  const vaultSecrets = await getVaultSecrets();

  // Use environment variables
  const config = {
    app: {
      port: process.env.APP_PORT,
      host: process.env.APP_HOST,
    },
    log: {
      level: process.env.LOG_LEVEL,
      pretty: process.env.LOG_PRETTY,
      redactions: ['config.db.connectionString'],
    },
    db: {
```

```
      connectionString: vaultSecrets.DATABASE_CONNECTION_STRING ??
      process.env.DATABASE_CONNECTION_STRING,
    },
    features: {
      exposeConfig: process.env.FEATURE_EXPOSE_CONFIG,
      allowedOrigins: process.env.ALLOWED_ORIGINS,
      useVault: process.env.FEATURE_USE_VAULT
    }
  }

  // Validate and parse configuration
  const validatedConfig = configSchema.parse(config);

  // Return immutable configuration
  return Object.freeze(validatedConfig);
}
```

Application (lib/app.js)

The application module demonstrates:

- **Feature Flags:** Conditional endpoint exposure based on configuration

```
import fastify from 'fastify';
import { fastifyPlugin } from 'fastify-plugin';
import { db } from '../plugins/db.js';

export async function createApp({ config, logger }) {
  const app = fastify({ loggerInstance: logger });

  // Register database plugin, pass config db subset
  app.register(fastifyPlugin(db), config.db);

  // Config endpoint, enabled by feature flag
  if (config.features.exposeConfig) {
    app.get('/config', async () => {
      return config;
    });
  }

  return app;
}
```

Database Plugin (plugins/db.js)

The database plugin shows:

- **Configuration Isolation:** Database-specific configuration is isolated
- **Runtime Validation:** Connection is validated at runtime

```
import postgres from 'postgres';

export async function db(app, options) {
  const pg = postgres(options.connectionString);

  // Validate connection
  try {
    await pg`SELECT 1`;
    app.log.info('Database connection successful');
  } catch (err) {
    app.log.fatal({ err }, 'Failed to connect to the database');
    throw new Error('DATABASE_CONNECTION_ERROR', { cause: err });
  }

  app.decorate('pg', pg);
  app.addHook('onClose', async () => {
    await pg.end();
  });
}
```

Logger (lib/logger.js)

The logger implementation features:

- **Sensitive Data Protection:** Redaction of sensitive information
- **Flexible Output:** Optional pretty printing for development
- **Custom Redaction:** Custom censoring of sensitive values

```
import pino from 'pino';

function censor(value) {
  if (typeof value === 'string') {
    return value ? `[redacted,len:${value.length}] ${value.
    substring(0, 3)}...` : value;
```

```
    }
    return '[redacted]';
  }

  export function createLogger(options) {
    let transport;
    if (options.pretty) {
      transport = { target: 'pino-pretty' };
    }

    return pino({
      level: options.level,
      redact: {
        paths: options.redactions,
        censor: censor
      },
      transport
    });
  }
```

Main Application (main.js)

```
async function main() {
  let config;
  try {
    config = await getConfig();
  } catch (err) {
    console.error(JSON.stringify({ err, message: 'Failed loading
    config' }, null, 2));
    process.exit(1);
  }

  let logger;
  try {
    logger = createLogger(config.log);
  } catch (err) {
    console.error(JSON.stringify({ err, message: 'Failed creating
    logger' }, null, 2));
    process.exit(1);
  }

  try {
    const app = await createApp({ config, logger });
    logger.info({ config }, 'App starting');
    await app.listen({ port: config.app.port, host: config.app.host
```

```
});
    logger.info({ port: config.app.port, host: config.app.host },
    'App started');
  } catch (err) {
    logger.fatal({ err }, 'App crashed on start');
    process.exit(1);
  }
}
```

Configuration Best Practices Demonstrated

- **Validation at Boundaries**
  Configuration is validated immediately when loaded
- **Type Safety**
  Strong typing through Zod schema
- **Environment-Based**
  Configuration through environment variables
- **Immutable Config**
  Prevents runtime configuration modifications
- **Modular Design**
  Each component receives only its required configuration
- **Secure Defaults**
  Sensible default values for optional settings
- **Feature Flags**
  Structured approach to feature management
- **Sensitive Data Handling**
  Built-in redaction of sensitive information
- **Clear Error Messages**
  Validation errors are clear and actionable

- **Single Source of Truth**
  Configuration is loaded once at startup

- **Secrets Management**
  Optional HashiCorp Vault integration for secure secrets

Environment Variables

Configure the application using these environment variables:

```
# Application
APP_PORT=3000
APP_HOST=localhost

# Logging
LOG_LEVEL=info
LOG_PRETTY=true

# Database
DATABASE_CONNECTION_STRING=postgres://user:pass@localhost:5432/db

# Features
FEATURE_EXPOSE_CONFIG=false
ALLOWED_ORIGINS=localhost:3000
FEATURE_USE_VAULT=false

# Vault Configuration (Optional)
VAULT_ADDR=http://127.0.0.1:8200
VAULT_ROLE=myapp
VAULT_SECRET_PATH=secret/data/myapp
SERVICE_ACCOUNT_TOKEN_PATH=/var/run/secrets/kubernetes.io/
serviceaccount/token
```

Vault Integration

The application supports HashiCorp Vault for secrets management:

1. Local Development:

- Set `FEATURE_USE_VAULT=false` to use environment variables
- Configure values directly in `.env` file

2. Kubernetes Deployment:

- Set `FEATURE_USE_VAULT=true` to enable vault integration
- Configure vault connection in deployment manifest:

```
env:
  - name: FEATURE_USE_VAULT
    value: "true"
  - name: VAULT_ADDR
    value: "http://vault:8200"
  - name: VAULT_ROLE
    value: "myapp"
  - name: VAULT_SECRET_PATH
    value: "secret/data/myapp"
```

- Service account token is automatically read from the default path

3. Vault Secret Format
   Store secrets in Vault using the same names as environment variables:

```
{
  "DATABASE_CONNECTION_STRING": "postgres://user:pass@db:5432/app",
}
```

4. Fallback Behavior:

- If a secret is not found in Vault, falls back to environment variables
- All configuration validation remains active regardless of source

Starting the application, we see the following output:

```
> node --env-file=.env main.js

[15: 11: 17.249]INFO(195842): App starting
config: {
  “app”: {
    “port”: 3000,
    “host”: “localhost”
  },
  “log”: {
    “level”: “info”,
    “pretty”: true,
    “redactions”: [
       “config.db.connectionString”
    ]
  },
  “db”: {
    “connectionString”: “[redacted,len:45] pos...”
  },
  “features”: {
    “exposeConfig”: false,
    “allowedOrigins”: [
      “localhost:3000”,
      “example.com”
    ],
    “useVault”: false
  }
}
[15: 11: 17.257] INFO(195842): Server listening at
http://127.0.0.1:3000
[15: 11: 17.257] INFO(195842): App started
port: 3000
host: “localhost”
```

In case of an error, the application will print the error message and exit with a non-zero exit code, for example with an empty DATABASE_CONNECTION_STRING and/or invalid APP_PORT:

```
> DATABASE_CONNECTION_STRING= APP_PORT = 80 npm start

  > final - example@1.0.0 start
    > node --env-file=.env main.js

{
  “err”: {
    “issues”: [
      {
        “code”: “too_small”,
        “minimum”: 1024,
        “type”: “number”,
        “inclusive”: true,
        “exact”: false,
        “message”: “Number must be greater than or equal to 1024”,
        “path”: [
          “app”,
          “port”
        ]
      },
      {
        “code”: “too_small”,
        “minimum”: 1,
        “type”: “string”,
        “inclusive”: true,
        “exact”: false,
        “message”: “String must contain at least 1 character(s)”,
        “path”: [
          “db”,
          “connectionString”
        ]
      }
    ],
      “name”: “ZodError”
  },
  “message”: “Failed loading config”
}
```

Wrapping Up

In conclusion, efficient configuration management is essential for developing scalable, maintainable, and secure applications. By implementing best practices such as centralizing configuration, validating settings, and securely handling secrets, we can establish a robust configuration system that easily adapts to various environments.

It is important to align configuration practices with the development, deployment, and branching strategies to maintain consistency and efficiency throughout the software lifecycle.

By following these best practices, you'll be able to reduce security risks, enhance the developer experience, and ensure that configuration changes do not introduce unintended issues, ultimately making your application more reliable and easier to manage as it evolves.

06

Structuring Large Applications

Core benefits of modularity, common architectural pitfalls & best practices for constructing robust and maintainable systems

06

Managing Configurations

Core benefits of modularity, common architectural pitfalls & best practices for constructing robust and maintainable systems

Node.js is fundamentally built around the concept of modularity. Each JavaScript file is treated as an independent module: a self-contained, reusable unit of code. A module typically encapsulates logic related to a specific context, which may include functions, classes, constants, and variables. By isolating related logic into modules, developers can decompose complex applications into smaller, focused components that are easier to reason about and maintain.

Each module defines and exposes a clear interface: usually through `module.exports` or `export statements:` allowing other parts of the application to interact with it in a controlled manner. This modular structure enhances code reusability, enforces separation of concerns, and contributes to better organization and long-term maintainability across the codebase.

As Node.js applications grow in size and complexity, effective module management becomes critical. Without a well-structured approach, large codebases can quickly become difficult to maintain, extend, or reason about. This chapter examines essential practices for organizing Node.js applications around a modular architecture: a strategy that facilitates clarity, testability, and long-term scalability.

We will explore the core benefits of modularity, identify common architectural pitfalls, and present best practices for constructing robust and maintainable systems. Topics will include: dependency injection as a means of decoupling components; the principle of separation of concerns to improve readability and testability; and alternative patterns to the traditional MVC architecture.

The ultimate goal is to equip you with the tools and mindset required to design Node.js applications that are clean, coherent, and prepared to evolve alongside your product and business.

6.1 What is a Module?

In Node.js, a module is a discrete unit of code organized in a single file or directory, designed to encapsulate related functionality.

Modules allow developers to isolate responsibilities, reduce duplication, and compose complex systems from simpler building blocks.

Each module typically exposes a well-defined interface: a set of exported functions, classes, constants, or configuration values that can be imported and reused elsewhere in the application.

Modules serve several important purposes:

Encapsulation
They hide implementation details and expose only what is necessary.

Reusability
Code can be reused across different parts of an application or even across projects.

Maintainability
Smaller, focused files are easier to test, debug, and extend.

Composability
Applications are built by composing many small, independent modules.

Node.js provides support for two primary module systems: CommonJS (cjs), which uses require and module.exports, and ECMAScript Modules (esm), which use import and export.

The following example adopts the esm syntax, which is the modern standard and offers backward compatibility for consuming cjs modules when needed.

auth.js

```
import jwt from 'jsonwebtoken';
import LRU from 'lru-cache';

const DEFAULT_EXPIRES_IN = '1h';
const DEFAULT_CACHE_MAX = 500;
const DEFAULT_CACHE_TTL = 1000 * 60 * 5;

const defaultCache = new LRU({
  max: DEFAULT_CACHE_MAX,
  ttl: DEFAULT_CACHE_TTL,
});

export function generateToken(user, { secret, expiresIn = DEFAULT_
EXPIRES_IN }) {
  return jwt.sign(
    { id: user.id, username: user.username },
    secret,
    { expiresIn }
  );
}

export function verifyToken(token, { secret, cache = defaultCache })
{
  const cached = cache.get(token);
  if (cached) return cached;

  try {
    const decoded = jwt.verify(token, secret);
    cache.set(token, decoded);
    return decoded;
  } catch {
    return null;
  }
}
```

As observed, only the two functions are exposed as part of the module's public interface, while all other elements remain encapsulated within the module's internal scope.

The Singleton Pattern: Convenience or Hidden Coupling?

The singleton pattern is a common technique used to ensure that a module exposes only a single, shared instance throughout the entire application. In Node.js, this is often achieved naturally due to the way modules are cached: once a module is loaded, its exports are cached, and subsequent `import` or `require` calls return the same instance.

This behavior can be convenient for utilities like loggers, configuration loaders, or database connections. However, overusing or misusing singletons can lead to unintended consequences, especially in large applications.

Real-World Example: A Logger Module

Consider a simple logging utility

`logger.js`

```
import pino from 'pino';

let logger;

export function createLogger(options) {
  if (logger) return logger;
  logger = pino(options);
  return logger
}
```

This is a classic singleton. Anywhere in the application where `logger.js` is imported, the same instance of the logger is used.

What's the Problem?

At first glance, this pattern seems harmless. But it becomes problematic when the application grows and requires:

- **Environment-specific configuration** (e.g., different log levels or formats)
- **Per-request context** (e.g., correlation IDs for tracing)
- **Testability** (e.g., capturing logs in tests or mocking the logger)

A singleton logger, instantiated once and reused across the entire application, makes these requirements challenging to fulfill. It behaves as an implicit global, sharing its configuration and state universally, which limits flexibility and introduces hidden coupling.

Shared Singletons and Module Resolution Pitfalls

The problem becomes particularly evident when a singleton is used as a shared dependency across multiple modules.

Due to how Node.js resolves modules, if multiple parts of the application (or multiple packages) depend on the same module and resolve the same version, Node.js will load it once, from the nearest `node_modules` directory, and cache that instance globally across the runtime.

This means that even if different parts of your application expect isolated instances, they will in fact be sharing the same singleton, unintentionally.

This can lead to unexpected behaviors, especially when the singleton holds internal state or context.

Example: Logger Loaded from Multiple Packages
Imagine you have the following structure:

```
myapp/
├── node_modules/
│   ├── logger/
│   ├── service-a/
│   └── service-b/
└── app.js
```

Let's say both `service-a` and `service-b` depend on a shared logger module, and both resolve to the same version of `logger`, as implemented above.
You might expect the following behavior:

- `service-a` gets its own isolated logger instance
- `service-b` gets a separate one, possibly with different configuration

But due to Node.js module resolution, only the first resolved module will be cached, and all subsequent imports will receive the same instance, even across seemingly isolated packages.

This means if `service-a` is loaded first and initializes the logger like this:

```
import { createLogger } from 'logger';

export function serviceA() {
  const logger = createLogger({ level: 'debug' });
  logger.info('Service A is running');
}
```

And then `service-b` later tries to use the logger with different options, for example a different log level:

```
import { createLogger } from 'logger';

export function serviceB() {
  const logger = createLogger({ level: 'warn' });
  logger.info('Service B is running');
}
```

Then in `myapp/app.js`

```
import { serviceA } from 'service-a';
import { serviceB } from 'service-b';

serviceA();
serviceB();
```

The expected behavior is to see only the message **"Service A is running"**, since its logger is configured at the `debug` level.

However, the output also includes **"Service B is running"**, even though Service B is intended to log only messages at the `warn` level or higher.

Here's what happens step by step:

1. **Step 1**
   Service A is loaded.
2. **Step 2**
   It calls createLogger with a debug log level.
3. **Step 3**
   The logger is created and returned.
4. **Step 4**
   The logger instance is cached by Node.js' module system.
5. **Step 5**
   Service B is loaded afterward.
6. **Step 6**
   It attempts to create a new logger with a warn level using createLogger.
7. **Step 7**
   Instead of getting a new logger, it receives the already cached logger instance from Service A.
8. **Step 8**
   Since the logger is still configured at debug level, Service B's info messages are printed — even though they should have been suppressed.

This can result in excessive logging, leaked configuration, and hard-to-debug side effects, especially in large monorepos or plugin-based systems.

The recommended solution to the issues caused by singletons is to adopt dependency injection. Rather than exporting stateful singletons from shared libraries or utility modules, design your components to receive their dependencies explicitly.

This approach allows each consumer to create and configure its own instance based on its specific context.

As a result, it preserves modularity, improves testability, and prevents unintended interactions between unrelated parts of the system.

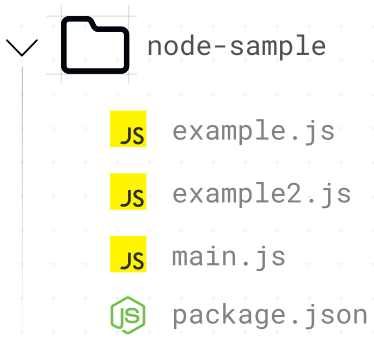
6.2 Dependency Injection

Dependency injection is a popular technique which makes producing independent and scalable modules easier.

In simpler terms, dependency injection is a pattern where instead of requiring or creating dependencies directly inside a module, they are passed as references or parameters.

There is a misconception that you need a framework to implement dependency injection, but this is incorrect: you can always use the depedency injection by constructor, as shown in the following example.

Say we have a Node project called node-sample with the folder structure on the side.



In the example.js file, we have

```
async function fancyCall() {
  console.log("Fancycall")
  return 42;
}

export async function buildGetToken() {
  let accessToken = null;

  const getToken = async () => {
    if (!accessToken) accessToken = await fancyCall();
    return accessToken;
  }

  async function stop() {
    accessToken = null;
    console.log("Access Token Stopped");
  }

  return {
    getToken,
    stop
  }
}
```

In the example.js file, we have

```
export async function buildDoSomething({ getToken }) {
  return {
    async something() {
      const token = await getToken();
      return token * 2;
    },

    async stop() {
      console.log("Something Stopped");
    },
  };
}
```

Then in the main.js we consume our dependencies

```
import { buildGetToken } from "./example.js";
import { buildDoSomething } from "./example2.js";

const { getToken, stop } = await buildGetToken();
const { something, stop: stop1 } = await buildDoSomething({ getToken });

console.log(await something());
console.log(await something());
console.log(await something());

await stop();
await stop1();
```

If we declare the `buildGetToken` function and the `buildDoSomething` functions in our main.js file, this means we have to modify the entire code every time we want to make changes to it.

Imagine this in a larger application– we'd have to search through our entire codebase to implement our changes. This would go against a common programming rule, **"Program to an interface, not an implementation."**

Moreover, dependencies make it easier to attach decorators to code. This forms the basis on which one of Fastify's major features is built: plugins.

Dependency injection also works well with classes. So if we were using classes, we would have this in the `example.js`.

```
async function fancyCall() {
  console.log("Fancycall")
  return 42;
}

class Example {
  constructor() {
    this.accessToken = null;
  }

  async getToken() {
    if (!this.accessToken) this.accessToken = await fancyCall();
    return this.accessToken
  }

  async stop() {
    this.accessToken = null;
    console.log("Access Token Stopped")
  }
}

export async function buildGetToken() {
  return new Example();
}
```

Then the example2.js file would look like this

```
class DoSomething {
  constructor({ example }) {
    this.example = example;
  }

  async something() {
    const token = await this.example.getToken();
    return token * 2;
  }

  async stop() {
    console.log("Something Stopped");
  }
}

export async function buildDoSomething({ example }) {
  return new DoSomething({ example });
}
```

In summary, dependency injection provides a lot of flexibility to our code, allowing us to create and implement multiple modules that are independent of each other.

6.3 Splitting Your Application Packages Into Modules

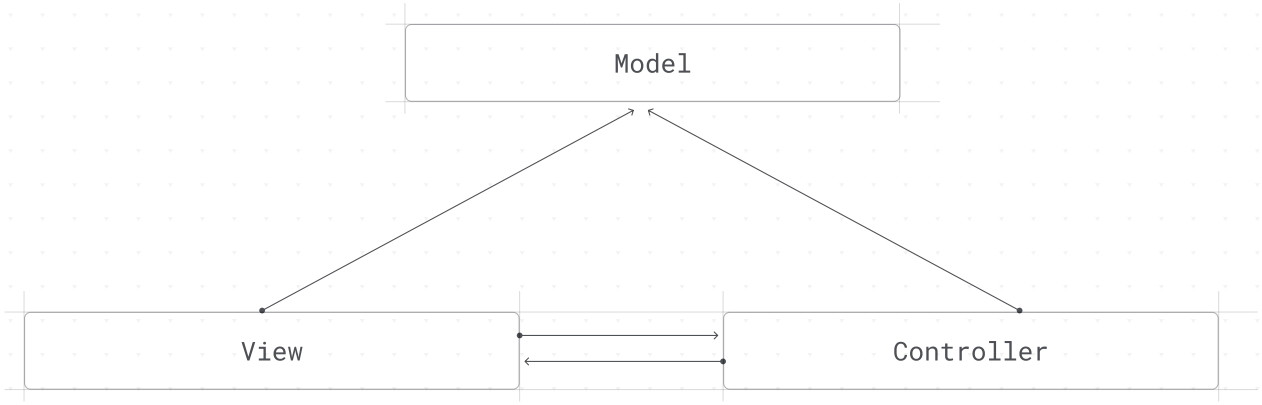
Most applications are built using the MVC (Model-View-Controller) model, whereby each section of your code has a specific and unique purpose, allowing you to split out the front and backend code into separate components.

As the name would suggest, this model is broken down into three parts: Model code, View code and Control code.

The Model is a central component, working with the database to hold raw-data, logic, and rules of an application.

The View is the user interface section of the app. In backend applications, it contains some HTML, CSS, XML, JS or other languages for the user interfaces. These user interfaces can be used to send basic forms to receive data to complete a process or update your users on a process.

The Controller handles all of the logic of the application. It handles all the processes and methods of the application, sends the response, and handles errors.



With MVC, the three core layers—Model, View, and Controller—define where new functionality can be introduced. While this separation of concerns provides structure, it can become a bottleneck in large-scale applications.

A major challenge with MVC is the inevitable growth in the number of files, classes, and dependencies, which can make code harder to navigate, debug, and refactor. Over time, the architecture can become rigid, forcing developers to modify multiple components when implementing changes. This increases the risk of regressions and makes feature development slower.

Another common issue is that MVC often leads to tight coupling between layers. Business logic frequently seeps into Controllers or even Views, leading to implicit dependencies that erode modularity. As a result, making changes in one part of the system can have unintended effects elsewhere, reducing maintainability.

While MVC works well for small to mid-sized applications, it struggles to scale cleanly.

So, what would be an alternative to the MVC model?

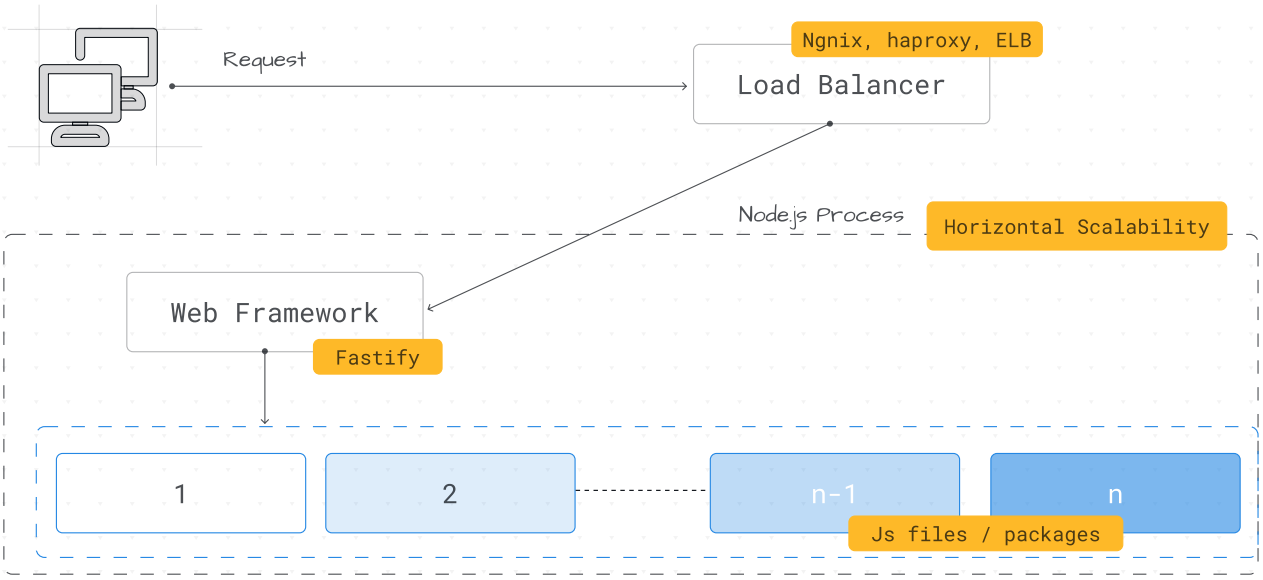
As complexity grows, modular architectures—such as service-based or domain-driven designs—can provide greater flexibility, better separation of concerns, and improved long-term maintainability.

Modules can help you scale your application's complexities, particularly with the One-Module-One-Feature approach. Under this approach, you would divide your applications into domain logic which spreads across your entire application.

Monolithic Application with One-Module-One-Feature Architecture

Let's see an example of a monolithic Node.js application designed following the **One-Module-One-Feature** principle: each feature or concern of the application is implemented in its own isolated module, encapsulating all relevant logic, dependencies, and interfaces.

This approach encourages separation of concerns, simplifies reasoning about the codebase, and improves maintainability even as the system grows in complexity.



The project is organized around two main types of modules:

1. Shared Plugins

These modules provide common infrastructure and cross-cutting functionality used across the application. They are initialized once, configured via dependency injection or environment variables, and passed to other modules as needed.

- `auth:` Handles authentication logic (e.g., JWT validation, session management).
- `db:` Provides access to the database, typically exposing a query interface or ORM client.
- `payment:` Integrates with payment providers (e.g., Stripe, PayPal) and handles payment flows.

Each of these modules is designed to be reusable, testable, and independent of application-specific logic. The number of shared modules in this example is intentionally limited for the sake of simplicity.

In a real-world application, however, this layer usually encompasses a broader range of concerns, including CORS handling, rate limiting, caching, security measures, and other cross-cutting functionality that are essential for building a robust and production-ready system.

2. Feature Modules (Routes)

These represent the **core business entities** of the ecommerce platform. Each feature module owns its own routing logic, domain-specific models, controllers, and service logic. They encapsulate everything related to a single domain concern:

- `users:` Manages user registration, login, profile updates, and authentication integration.
- `products:` Handles product catalog, inventory management, and product metadata.
- `orders:` Manages order creation, order history, status updates, and fulfillment logic.

Each route module typically exports a router as a fastify plugin that will be registered in the main application, and receives injected dependencies (such as logger, db, or auth) for maximum flexibility and testability.

The lib/ directory contains shared stateless utilities that are generic and horizontal to the application, such as configuration loaders, custom error classes, or validation helpers.

Structure

```
/ecommerce
├── src/
    ├── lib/
    │   ├── config.js
    │   ├── plugins/
    │   ├── auth/
    │   ├── db/
    │   ├── ...
    │   ├── payment/
    ├── routes/
    │   ├── users/
    │   ├── products/
    │   ├── ...
    │   ├── orders/
    ├── app.js
    ├── server.js
```

- `plugins/` contains shared infrastructure.
- `routes/` contains feature modules, each handling one business domain.
- `app.js` initializes the shared plugins and assembles all modules.
- `main.js` starts the HTTP server and bootstraps the application.

We'll walk through the code involved in setting up the application and processing an order with payment.

This example will give us a practical look at how the key components of the system come together in a real scenario.

---

src/app.js

The app.js file serves as the composition root of the application — the place where all parts of the system come together.

Its primary responsibility is to instantiate the Fastify server, register shared plugins (such as logging, configuration, authentication, and database access), and mount feature modules that expose the business logic.

```
import Fastify from 'fastify';

// Shared plugins
import db from './plugins/db/index.js';
import auth from './plugins/auth/index.js';
import payment from './plugins/payment/index.js';

// Feature modules (routes)
import ordersRoutes from './routes/orders/index.js';
import productsRoutes from './routes/products/index.js';
import usersRoutes from './routes/users/index.js';

export async function buildApp({ config }) {
  const fastify = Fastify(config.fastify.options);

  // Register shared plugins
  fastify.register(db);
  fastify.register(auth);
  fastify.register(payment);

  // Register feature modules
  fastify.register(usersRoutes, { prefix: config.routes.users });
  fastify.register(productsRoutes, { prefix: config.routes.products });
  fastify.register(ordersRoutes, { prefix: config.routes.orders });

  return fastify;
}
```

---

src/main.js

The `main.js` file is the entry point of the application. It bootstraps the Fastify server by calling the `buildApp` function from `app.js`, starts listening on the configured port, and handles startup errors gracefully.

This file is minimal by design, keeping all initialization logic centralized in app.js for better separation of concerns.

```
import { buildApp } from './app.js';
import { config } from './lib/config.js';

async function start() {
  const app = await buildApp({ config });

  try {
    await app.listen({ port: config.app.port, host: config.app.host });
    app.log.info('Application started');
  } catch (err) {
    app.log.fatal({ err }, 'Unable to start the application');
    process.exit(1);
  }
};

start();
```

---

src/routes/orders/index.js

The `orders/index.js` file defines the `/orders/complete endpoint`, which handles the finalization of an order. The route is authenticated, validates input, and wraps the entire process in a database transaction.

It charges the customer using the payment plugin and updates the order status to `completed` if successful. This implementation demonstrates clean integration between feature logic and shared infrastructure.

```
export default async function ordersRoutes(fastify, opts) {
  const { db, payment, logger } = fastify;

  fastify.post('/complete', {
    preHandler: fastify.auth, // assumes fastify.auth sets request.user
    schema: {
      body: {
        type: 'object',
        required: ['orderId', 'paymentMethod'],
        properties: {
          orderId: { type: 'string' },
          paymentMethod: { type: 'string' },
        },
      },
    },
    handler: async (request, reply) => {
```

```
      const { orderId, paymentMethod } = request.body;
      const userId = request.user.id;

      const trx = await db.beginTransaction();

      try {
        const order = await db.getOrderById(orderId, trx);
        if (!order || order.userId !== userId) {
          await trx.rollback();
          return reply.code(404).send({ error: 'Order not found' });
        }

        if (order.status !== 'pending') {
          await trx.rollback();
          return reply.code(400).send({ error: 'Order cannot be
          completed' });
        }

        const result = await payment.charge({
          amount: order.totalAmount,
          method: paymentMethod,
          metadata: { orderId, userId },
        });

        if (!result.success) {
          await trx.rollback();
          return reply.code(402).send({ error: 'Payment failed' });
        }

        await db.updateOrderStatus(orderId, 'completed', trx);
        await trx.commit();

        logger.info({ orderId, userId }, 'Order completed');
        return reply.send({ status: 'completed', transactionId:
        result.transactionId });

      } catch (err) {
        await trx.rollback();
        logger.error({ err, orderId, userId }, 'Order completion
        failed');
        return reply.code(500).send({ error: 'Internal server error' });
      }
    },
  });
}
```

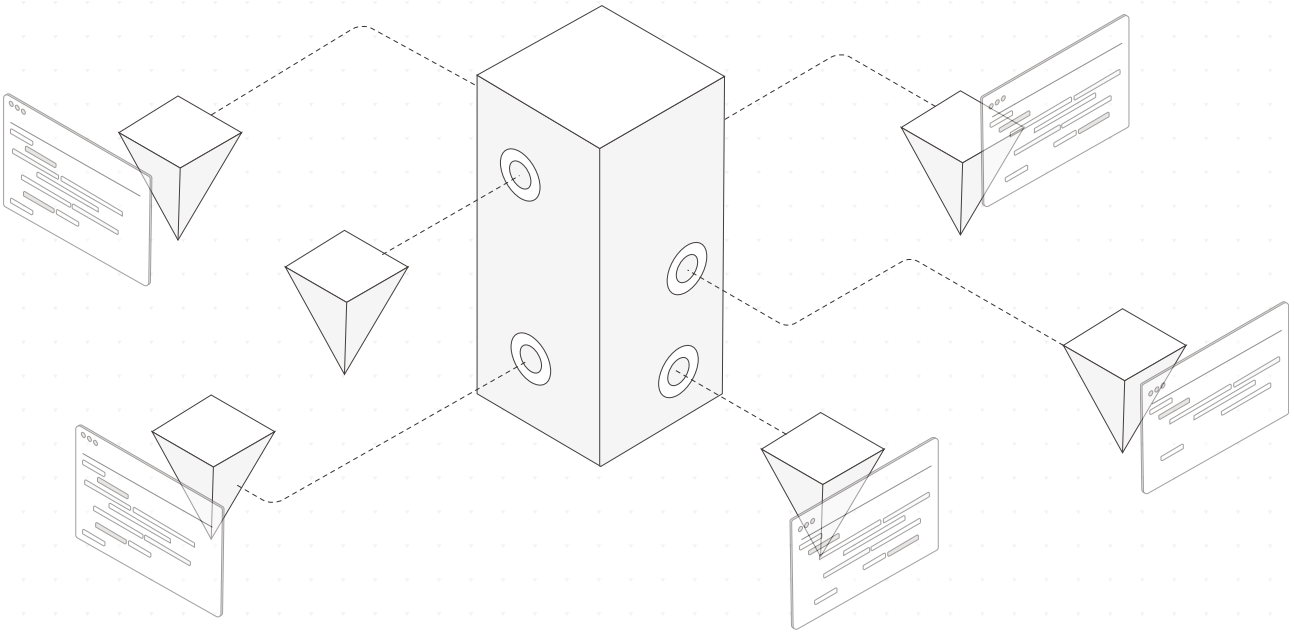
This architectural approach offers several key advantages that make it well-suited for building robust and maintainable systems. By establishing clear boundaries between infrastructure and business logic, it ensures that each concern remains isolated, reducing complexity and making the codebase easier to reason about. Modules are designed with high cohesion and minimal coupling, which promotes internal consistency while allowing different parts of the system to evolve independently.

One of the major strengths of this structure is its scalability. Although it begins as a monolith, the architecture naturally supports a transition to a modular monolith or even a microservices-based system as the application grows. This gradual evolution helps teams avoid premature complexity while keeping future options open.

Testability is another important benefit. Since modules can be developed and executed in isolation, with external dependencies mocked or stubbed, testing becomes more straightforward and reliable.

This also contributes to a smoother onboarding experience: new developers can focus on understanding and contributing to a single module at a time, without needing to comprehend the entire codebase from day one.

Overall, this architecture lays down a clean and scalable foundation for complex applications, without the initial overhead of distributed systems. It is particularly effective for startups or teams that prioritize development speed, clarity, and simplicity, while still preserving the flexibility to refactor or extract services as the product evolves.



6.4 From Monolith to Microservices: Evolving with Growth

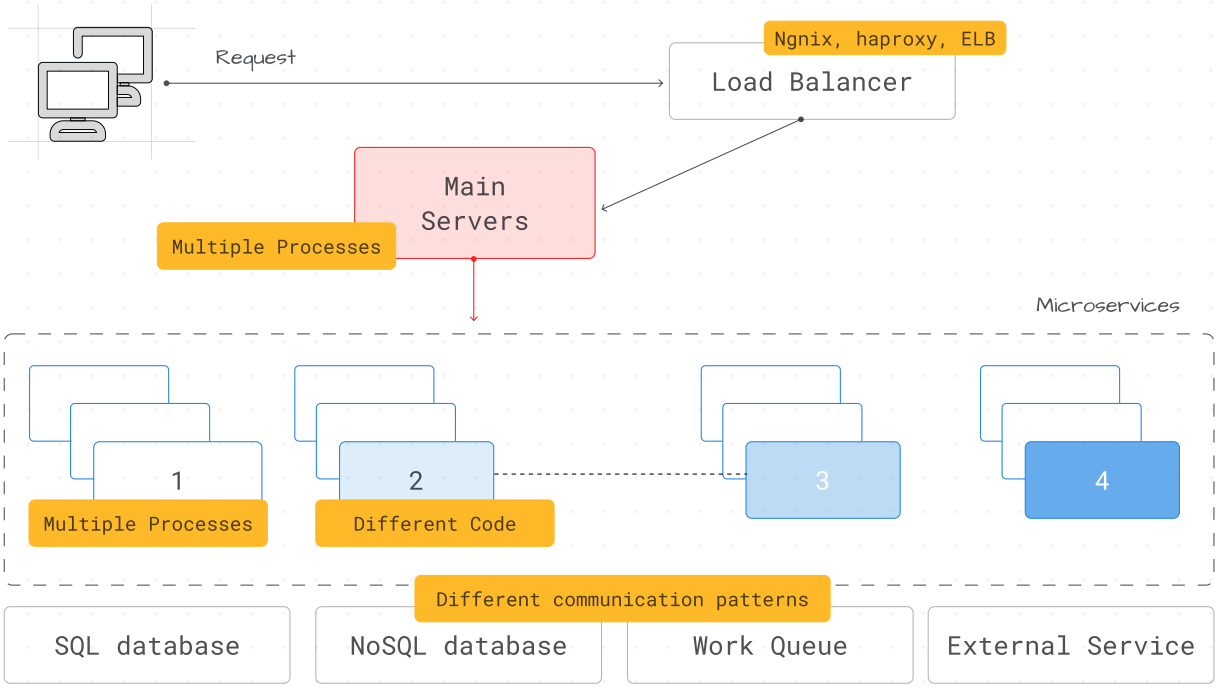
A well-structured monolithic application provides a solid foundation for product development, especially in the early stages of a business. When built with clear boundaries, modular design, and proper separation of concerns—such as through the One-Module-One-Feature approach—it remains maintainable, scalable, and easy to evolve.

As the business grows, however, the application often needs to accommodate a broader set of features, serve a higher volume of users, and meet increasingly demanding non-functional requirements such as scalability, fault isolation, team autonomy, and faster deployment cycles.

At this stage, a cleanly designed monolith becomes an advantage rather than a liability. Thanks to its modular structure and clear interfaces between features and services, it becomes much easier to extract specific modules and evolve them into independent microservices.

For example, components like payment processing, order fulfillment, or inventory management—if already encapsulated in their own feature modules—can be gradually moved out of the monolith. These services can then be deployed, scaled, and versioned independently, without requiring a complete rewrite of the system.

This transition allows organizations to adopt a microservices architecture incrementally, in a controlled and sustainable way. Instead of being driven by premature abstraction or complexity, the move to microservices is informed by real-world needs—scaling bottlenecks, organizational scaling, or domain ownership. In other words, a thoughtfully designed monolith acts as a stepping stone to distributed architecture, enabling teams to evolve the system at their own pace, with minimal risk and maximum clarity.



After building a well-structured monolithic application following the one-module-one-feature approach—with distinct modules for users, orders, and products, and shared fastify plugins such as authentication, database access, and payment processing—we are now ready to explore how to evolve this design into a microservices architecture.

In this new setup, each feature becomes its own standalone service. The users, orders, and products modules are extracted into separate applications, each running independently and managing its own database. This separation enables better scalability, isolation, and deployment flexibility.

At this stage, we adopt a monorepo and deliberately avoid introducing asynchronous communication patterns or event-driven architecture. Instead, we focus on simplicity and direct service-to-service interaction, prioritizing code design over the complexity of transport mechanisms.

```
/ecommerce-monorepo
├── services/
│   ├── users/
│   │   ├── src/
│   │   ├── lib/
│   │   │   ├── config.js
│   │   ├── plugins/
│   │   │   ├── auth.js
│   │   ├── routes/
│   │   │   ├── auth.js
│   │   │   ├── profile.js
│   │   ├── app.js
│   │   ├── main.js
│   ├── orders/
│   │   ├── src/
│   │   │   ├── lib/
│   │   │   │   ├── config.js
│   │   │   ├── plugins/
│   │   │   │   ├── auth.js
│   │   │   │   ├── db.js
│   │   │   │   ├── payment.js
│   │   │   ├── routes/
│   │   │   │   ├── orders.js
│   │   │   ├── app.js
│   │   │   ├── main.js
│   ├── products/
│   │   ├── src/
│   │   │   ├── lib/
│   │   │   │   ├── config.js
│   │   │   ├── plugins/
│   │   │   │   ├── auth.js
│   │   │   ├── routes/
│   │   │   │   ├── products.js
│   │   │  app.js
│   │   │  main.js
```

In this microservice architecture, each application runs its own web server tailored to its specific business requirements. Initially, since they originate from a monolithic split, these services tend to share a similar structure. However, over time they naturally diverge as they evolve to meet distinct business needs and handle varying workloads.

Because of this divergence, sharing code across services—even for common concerns like logging, telemetry, or configuration—is often not practical. Reuse may introduce tight coupling or unnecessary complexity, hindering independent evolution and scalability.

6.5 Solving Microservice Challenges

Adopting a microservice architecture offers significant benefits—but it also introduces challenges related to coordination and cost.

Microservices are an architectural pattern in which each component operates independently. This enhances flexibility, scalability, and fault isolation, making the approach attractive for large teams building systems at scale. By allowing teams to work autonomously on discrete services, microservices promote faster development cycles and easier maintenance.

However, this autonomy can also lead to new problems.

One common issue is **coordination overhead**. With multiple independent teams managing different services, aligning on shared goals or architectural decisions can slow down progress. Cross-team communication becomes essential, yet more difficult.

Another major concern is **infrastructure cost**. Running a growing number of services—each with its own resource requirements—can lead to significant overhead if not carefully managed. Efficient resource allocation, monitoring, and orchestration become critical to keeping costs under control.

Finally, while breaking down a Node.js application into independent modules or services can improve modularity and performance, it also introduces **deployment complexity.** Each service must be deployed, monitored, and maintained independently, adding operational overhead.

At this stage, we begin to separate concerns by organizing the system into distinct microservices, each responsible for a specific domain and encapsulating its own business logic. This distributed architecture establishes clear boundaries between services, allowing teams to work independently and enabling the system to scale more predictably.

Even with a minimal setup, these modular services can handle thousands of users and requests per second—providing scalability and resilience well-suited to the needs of a growing startup.

However, it may still be premature to fully split the codebase into independently deployed services. Instead, these services can continue to run within the same monorepo or process space. This approach avoids the early overhead of inter-service networking, separate deployments, and service discovery—while still reaping the benefits of modularity, clear domain separation, and parallel development.

This hybrid model captures many of the advantages of microservices—such as isolation, maintainability, and scalability—without the immediate cost of full operational complexity. The first major hurdle in a true microservices setup is deployment: managing multiple services, containers, versions, and orchestrators can quickly become a heavy operational burden.

6.6 Modular Monolith

At this point, one thing becomes clear: **microservices may not be the right fit for most development teams**, especially in the early stages of a product's lifecycle. The architectural complexity, operational overhead, and coordination demands can outweigh the benefits—particularly when the team is small or the domain is still evolving.

This realization leads to a more pragmatic alternative: the **Modular Monolith**.

A Modular Monolith retains the internal structure and boundaries of a microservice architecture—clear separation of domains, encapsulated business logic, and modular components—but runs everything within a single deployable unit. It allows teams to enjoy the organizational and maintainability benefits of modular design, without the cost of distributed systems. It's an approach that scales well with the team and the product, providing a clear path toward microservices **only when necessary**, not by default.

Multithreading provides a practical solution to the challenges of scaling modular monoliths by enabling services to run independently while sharing system resources. In Node.js, the `worker_threads` module allows developers to execute JavaScript code in parallel within the same process. This makes it possible to distribute workloads more efficiently and fully utilize available CPU cores.

However, multithreading introduces its own set of complexities. Since worker threads share memory, careful coordination is required to avoid contention and ensure proper resource isolation. A failure in one thread can potentially affect others if not properly sandboxed. Additionally, developers must manage the lifecycle of threads—spawning, monitoring, and restarting them as needed—adding orchestration overhead to the application logic.

Platformatic addresses these challenges by simplifying the implementation of networkless HTTP communication within a modular monolith architecture. It abstracts the underlying complexity of worker thread orchestration and inter-service communication, making it easier to build scalable, high-performance Node.js applications without prematurely adopting a full microservices architecture.

6.7 Introducing Platformatic Watt

Watt is the Node.js application server.

It enables you to run multiple Node.js services that are centrally managed and coordinated, streamlining the orchestration of complex applications. Services can be executed using **worker threads**, which offer faster startup times and lower overhead, or as **child processes**, which are better suited for services with complex startup sequences or that require greater isolation.

Watt allows you to run multiple Node.js applications under one roof, with centralized control and configuration.

It comes **production-ready out of the box**, equipped with a rich set of features that every service typically needs: **inter-service networking**, support for `.env` **file loading**, comprehensive **logging**, and **distributed tracing**.

All of these features can be easily configured using simple **JSON file**s, eliminating boilerplate code and improving long-term maintainability of your applications.

Watt Key Features

Automatic Multithreading
Watt intelligently parallelizes your services across available CPU cores with a single command. This built-in multithreading improves performance and resource efficiency—no manual thread orchestration required.

Comprehensive Non-Functional Requirements (NFR) Management
Out of the box, Watt handles essential cross-cutting concerns such as logging, tracing, and resource coordination. It abstracts the complexity of managing these non-functional requirements, ensuring consistent and reliable behavior across all services.

Integrated OpenTelemetry Tracing
With built-in support for OpenTelemetry, Watt provides real-time observability into your distributed system. You can trace requests across services, pinpoint performance bottlenecks, and gain deep insights—without any additional configuration.

Unified Logging with Pino

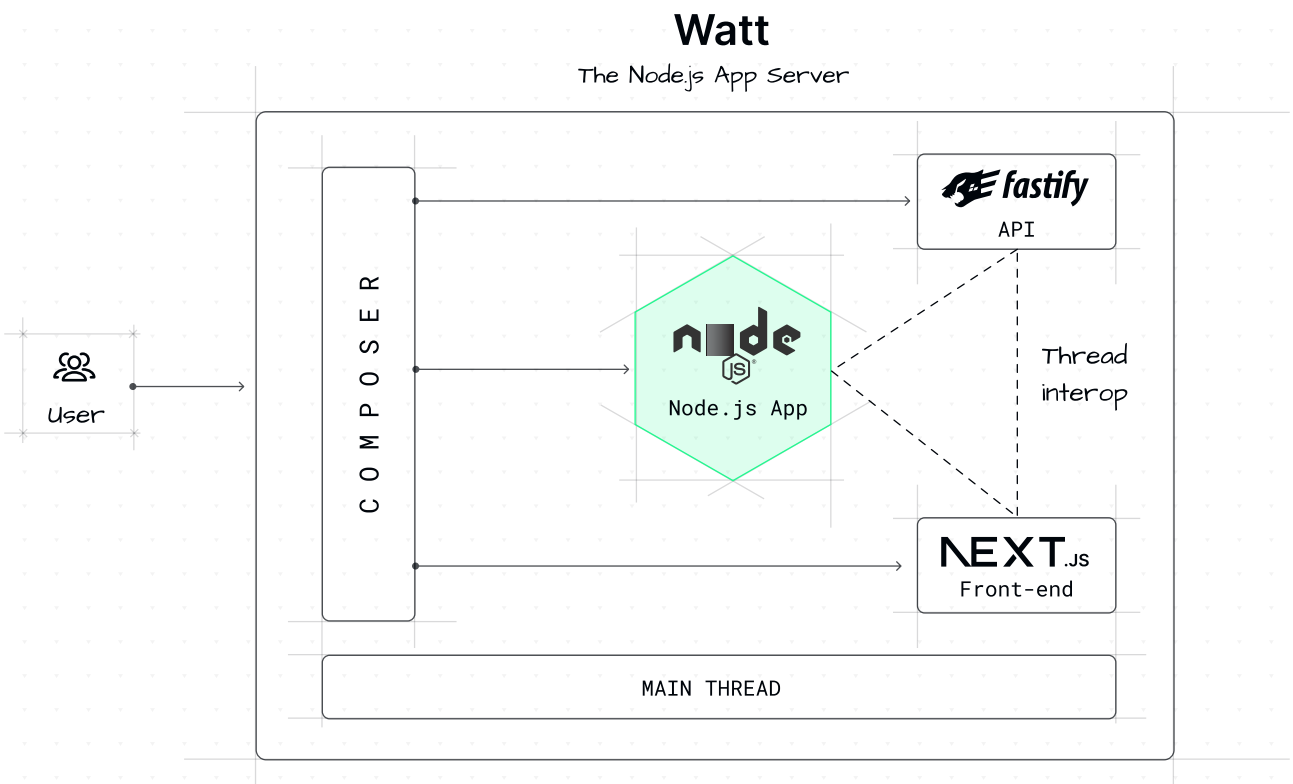
Watt leverages Pino for high-performance, structured logging. It delivers consistent and centralized logs across all services, making it easier to monitor performance, troubleshoot issues, and maintain visibility across your Node.js architecture.

This flexibility makes Watt ideal for building scalable, modular architectures where different services have different runtime requirements. It serves as a solid foundation for modern Node.js platforms that prioritize modularity, performance, and operational clarity. By removing deployment complexity and offering powerful built-in features, Watt delivers a streamlined, productive developer experience.

Additionally, Watt includes built-in health check routes, making it ready for seamless integration with Kubernetes (k8s) and other orchestration systems right out of the box.

Watt as Modular Monolith framework

This is where Platformatic Watt simplifies the process.



Instead of manually managing worker threads, Watt allows developers to run multiple services within a single Node.js process, with each service executing in its worker thread.

This design helps optimize CPU usage while keeping services modular and isolated. Watt reduces infrastructure overhead while maintaining performance by eliminating the need to run each microservice as a separate deployment unit (or container).

For example, consider an application with an API gateway, authentication, and data processing services. In a traditional setup, each of these would run as a separate deployment, leading to organizational inefficiencies.

With Watt, these services can run concurrently in separate worker threads within the same process, preserving modularity without unnecessary overhead.

Fault isolation ensures that if one service encounters an issue, it doesn't impact the others. By fully using available CPU cores and automating much of the thread management, Watt provides a structured and scalable way to integrate multithreading into a Node.js application without adding unnecessary complexity.

By leveraging multithreading intelligently, large applications can remain performant without becoming a maintenance nightmare.

Platformatic Watt provides a way to harness the benefits of worker threads while keeping the architecture simple and scalable.

In-Process Mesh Network

Watt enhances the services it manages by providing an **internal communication system** that enables **networkless HTTP calls** when services are running as worker threads.

This eliminates the overhead typically associated with network-based communication, resulting in faster, more efficient service interactions.

Beyond performance, this approach also improves **security**: since data is exchanged entirely within the process memory, there is no network surface to secure, reducing the risk of interception or external attacks.

Modern Tools for Implementing Networkless HTTP

Let's dive a little deeper into how **networkless HTTP** can be implemented in Node.js applications using modern tools—specifically, **Undici** and **Fastify**.

Undici

The native HTTP stack in Node.js has known performance limitations that are difficult to resolve without breaking backward compatibility. This was the motivation behind **Undici**, a fast, modern HTTP library for Node.js. It powers the built-in `fetch` implementation in Node.js core and has become the go-to solution for high-performance HTTP communication.

Originally, Undici supported only HTTP/1.1, but it has since evolved to include support for HTTP/2, making it compatible with the latest versions of the HTTP protocol. One of its core strengths lies in its dispatcher interface, which simplifies the complexity of making HTTP calls at scale. Developers define how requests are handled using the `dispatch` method, providing fine-grained control over request execution.

Fastify

Fastify, on the other hand, provides a powerful feature for performing networkless calls through its `.inject()` method. This method allows a request to be simulated as if it came from the network, while actually being processed entirely in-memory. The Fastify server performs all routing, validation, and response handling internally—without needing a real network layer.

Undici + Fastify = Internal HTTP Efficiency

By leveraging Fastify's `.inject()` mechanism and Undici's efficient dispatcher interface, it's possible to build fully networkless HTTP communication within a Node.js process. This approach maintains the same familiar HTTP interface while optimizing performance and reducing overhead.

Watt: Bringing It All Together

Watt builds on top of these capabilities to provide internal HTTP networking out of the box. By combining Undici and Fastify under the hood, Watt enables services running as threads to communicate using efficient, networkless HTTP.

This design delivers maximum performance, reduces latency, and significantly enhances the developer experience, all without sacrificing the simplicity of traditional HTTP service interfaces.

Using Watt

Let's explore how to implement a modular monolith architecture using Watt, and how to build scalable applications with minimal boilerplate by leveraging Platformatic's built-in capabilities.

With Watt, you can structure your Node.js applications into cleanly separated services while running them efficiently in a single runtime. This approach gives you the benefits of modularity and isolation without the operational overhead of full microservices.

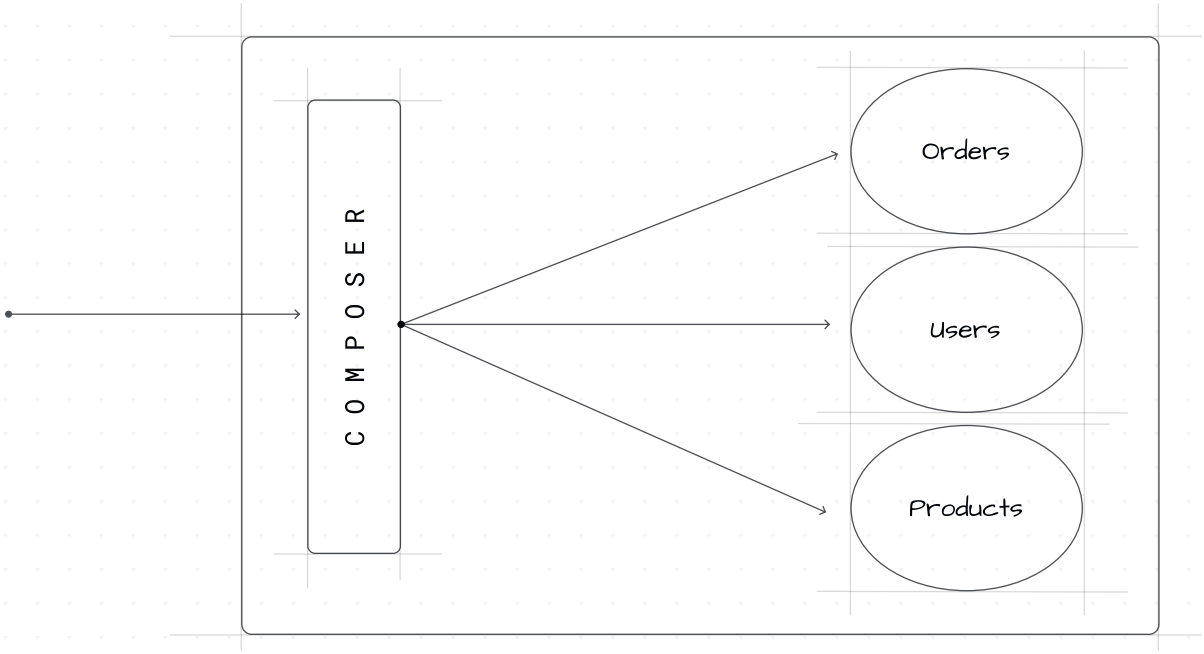
Project Structure

Here's a sample directory layout for a Watt-powered project:

```
watt/
├── .env
├── watt.json
├── pnpm-workspace.yaml
├── services/
│   ├── composer/
│   │   ├── platformatic.json
│   ├── orders/
│       ├── platformatic.json
│       ├── plugins/
│       │   ├── auth.js
│       │   ├── config.js
│       │   ├── db.js
│       │   ├── payment.js
│       ├── routes/
├── orders.js
```

This structure embraces a monorepo layout with multiple services under a single codebase, each with its own configuration and domain logic. The services are automatically detected and orchestrated by Watt, minimizing setup and maximizing modularity.

Each service can define its own routes, plugins, and configuration—yet run efficiently as part of the same process or runtime, thanks to Watt's smart threading model and internal HTTP mesh.



watt.json

This is a typical Watt runtime configuration.

```
{
  “$schema”: “https://schemas.platformatic.dev/@platformatic/
  runtime/2.55.0.json”,
    “entrypoint”: “composer”,
    “watch”: true,
    “basePath”: “{PLT_BASE_PATH}”,
    “autoload”: {
      “path”: “services”
    },
    “server”: {
      “hostname”: “{PLT_HOSTNAME}”,
      “port”: “{PLT_PORT}”
    },
    “logger”: {
      “level”: “{PLT_LOGGER_LEVEL}”
    }
}
```

services/composer/platformatic.json

This is the Composer configuration, where the service map is defined.

```
{
  “$schema”: “https://schemas.platformatic.dev/@platformatic/
  composer/2.55.0.json”,
    “server”: {
    “healthCheck”: true
  },
  “composer”: {
    “services”: [
      {
        “id”: “orders”
      },
      {
        “id”: “products”
      },
      {
        “id”: “users”
      }
    ],
      “refreshTimeout”: 1000
  },
  “watch”: true
}
```

`services/orders/platformatic.json`

This is a service configuration, where we define plugins and routes, in this case the orders service. Routes and plugins follow the same pattern and implementation as in monoliths and microservices, and that's all you need to implement for each service.

```
{
  "$schema": "https://schemas.platformatic.dev/@platformatic/
  service/2.55.0.json",
    "service": {
    "openapi": true
  },
  "watch": true,
    "plugins": {
    "paths": [
      {
        "path": "./plugins/config.js"
      },
      {
        "path": "./plugins/auth.js"
      },
      {
        "path": "./plugins/db.js"
      },
      {
        "path": "./plugins/payment.js"
      },
      {
        "path": "./routes/orders.js",
        "routePrefix": "/orders"
      }
    ]
  }
}
```

Scaffolding and Automation

Platformatic Watt provides an opinionated project structure that minimizes boilerplate and lets you focus on building your business logic.

Using the `create-platformatic` CLI tool, the project is auto-generated following a convention-over-configuration approach, reducing setup time and enforcing best practices.

`watt.json:` Central Runtime Configuration

The main configuration file, `watt.json`, defines how Watt runs and orchestrates your services. It automatically:
Sets the entry point of the application (typically the Composer service)

- Configures auto-loading of services from the services/ directory
- Establishes environment variable handling
- Enables built-in structured logging

Each individual service includes its own `platformatic.json`, where you declare its plugins, routes, and specific configuration.

The Composer Role

The Composer service acts as the orchestration layer of your application. It plays a central role in managing the lifecycle and communication of your services:

- Serves as the entry point for the entire application (as defined in `watt.json`)
- Automatically discovers and manages services within the system
- Acts as a unified API gateway, exposing a central HTTP interface
- Handles service registration, health checks, and routing
- Enables seamless service-to-service communication

The Composer's behavior is defined in `services/composer/platformatic.json`, where you configure:

- The list of managed services
- Refresh timeouts for dynamic service discovery
- Health check settings for Kubernetes or other orchestrators

Environment Variables

Watt streamlines environment management by automatically loading variables from a `.env` file and exposing them across all services:

- Global variables, defined in `watt.json`, are shared across every service
- Service-specific variables can be declared in each service's own configuration

This unified approach eliminates the need for manual `.env` handling in each service, simplifying configuration management and improving consistency across environments.

Built-in Functionalities

Watt offers a rich set of built-in capabilities out of the box, significantly reducing the need for custom infrastructure code and boilerplate.

Logging

- Structured JSON logging with configurable log levels
- Uniform logging format across all services
- Context-aware request logging for easier debugging and tracing

OpenTelemetry (Distributed Tracing)

- Built-in support for distributed tracing
- Automatic instrumentation of incoming and outgoing HTTP requests
- Performance metrics collection for real-time observability

OpenAPI Documentation

- Automatic generation of API documentation
- Built-in support for interactive Swagger UI
- Schema validation for incoming requests

Health Checks

- Built-in service health monitoring
- Automatic health check endpoints for integration with orchestration tools like Kubernetes

Hot Reloading

- File watching in development mode
- Automatic service restarts on code changes

Advantages Over Traditional Approaches

Compared to traditional monoliths or manually configured microservices, Watt offers a number of key benefits:

- Consistent architecture across services
- Centralized and declarative configuration
- Automatic service discovery
- Built-in observability and monitoring
- Simplified deployment (including Docker support)
- Reduced maintenance burden and boilerplate code

Wrapping Up

By embracing a modular approach to development in Node.js, you gain a powerful tool for crafting well-structured, maintainable, and scalable applications. Modular code promotes reusability, reduces redundancy, and simplifies the process of collaboration and code sharing. This chapter explored key concepts like dependency injection and alternative strategies to MVC architecture, all of which contribute to building robust and adaptable Node.js applications.

However, modularity alone isn't enough as applications scale—performance bottlenecks can emerge, particularly in large, resource-intensive workloads. This is where Platformatic Watt can help.

Watt improves Node.js scalability by executing multiple services as worker threads within the same process, addressing the traditional single-threaded constraints of Node.js. By allowing services to run in separate worker threads, Watt enhances CPU utilization, reduces over-provisioning, and lowers infrastructure costs, all while maintaining the simplicity of a modular architecture. This makes it particularly valuable for developers looking to scale applications efficiently without introducing unnecessary complexity.

By integrating Watt into a Node.js project, developers can take full advantage of modular design without sacrificing performance. Instead of relying on complex clustering or external load balancing, Watt enables a more streamlined, resource-efficient execution model that complements the modular principles discussed in this chapter.

By adhering to the best practices outlined in this chapter—including responsible module management, and leveraging performance-enhancing solutions like Watt—you can significantly mitigate risks while optimizing scalability. Together, these practices empower you to tackle complex projects with confidence, ensuring your Node.js code remains both maintainable and high-performing.

07

Running Node.js in the Cloud

A deep dive into deployment options and optimization techniques for Node.js

07

Running Node.js in the Cloud

A deep dive into deployment options and optimization techniques for Node.js

Running Node.js applications in the cloud can provide several advantages over conventional deployments, including standardization, improved performance, and monitoring.

When evaluating different cloud-based deployments, use the following criteria:

Scalability
Does the platform provide different scaling solutions, like vertical and horizontal, based on runtime data?

Elasticity
Will resources be automatically increased to meet demand, without intervention?

Cost-efficiency
Is the solution pay-as-you-go, increasing based on use, or a flat fee?

Reliability
What are the reliability guarantees, and how can they be ensured?

This chapter explores running Node.js in the cloud, the costs and benefits of different deployment solutions, and optimization techniques for Node.js.

7.1 Node.js and Docker: The First Step to Cloud Deployments

Why Containerization Matters

Docker is a containerization platform that allows developers to package applications and their dependencies into isolated, portable containers. This eliminates the "works on my machine" problem by ensuring consistency across development, testing, and production environments.

By using Docker, a Node.js application can be packaged with all required dependencies, configurations, and runtime settings, ensuring consistent execution regardless of the underlying infrastructure.

Building a Dockerfile for a Node.js Application

General Best Practices for Dockerizing Node.js Applications

When Dockerizing a Node.js application, the primary goals are to create minimal, secure, and reproducible images that are both fast to build and efficient to run. To achieve this, consider the following best practices:

Use official minimal base images, such as `node:22-slim or node:22-alpine`, to reduce both image size and potential attack surface.

Leverage multi-stage builds to cleanly separate build-time dependencies (like TypeScript, native compilers, or dependencies buildings) from the final runtime image. This keeps the production image smaller and free of unnecessary tools.

Install only production dependencies when no build step is needed. Set `NODE_ENV=production and` use `npm ci --only=production (or --omit=dev` with newer npm versions) to exclude development packages.

Avoid running as root in the final image. Create a non-privileged user and switch to it using `USER`, which greatly improves container security.

Use a `.dockerignore` file to exclude unnecessary files and directories (such as `node_modules`, `.git`, `tests`, and documentation) from being copied into the image, as well as leaking secrets.

This reduces context size and prevents accidental bloating, here an example:

```
# Exclude unnecessary dirs
node_modules/
test/
doc/

# Ignore any potential secrets
.env *

# Do not include development files
Dockerfile
.git/
.dockerignore
.gitignore
```

Pin your dependencies to ensure reproducible builds. Lock both the Node.js base image version (e.g. `node:22-slim`) and your npm dependencies via `package-lock.json`; `package-lock.json` should be committed on the repository.

Cache layers effectively. Always copy and install dependencies before copying application code to take advantage of Docker layer caching and speed up rebuilds when only code changes.

Use precompiled binaries or tools like `prebuildify` when working with native Node.js modules. This avoids compiling dependencies during runtime and ensures compatibility with the production environment.

Let's now look at different Dockerization strategies based on the nature of the application.

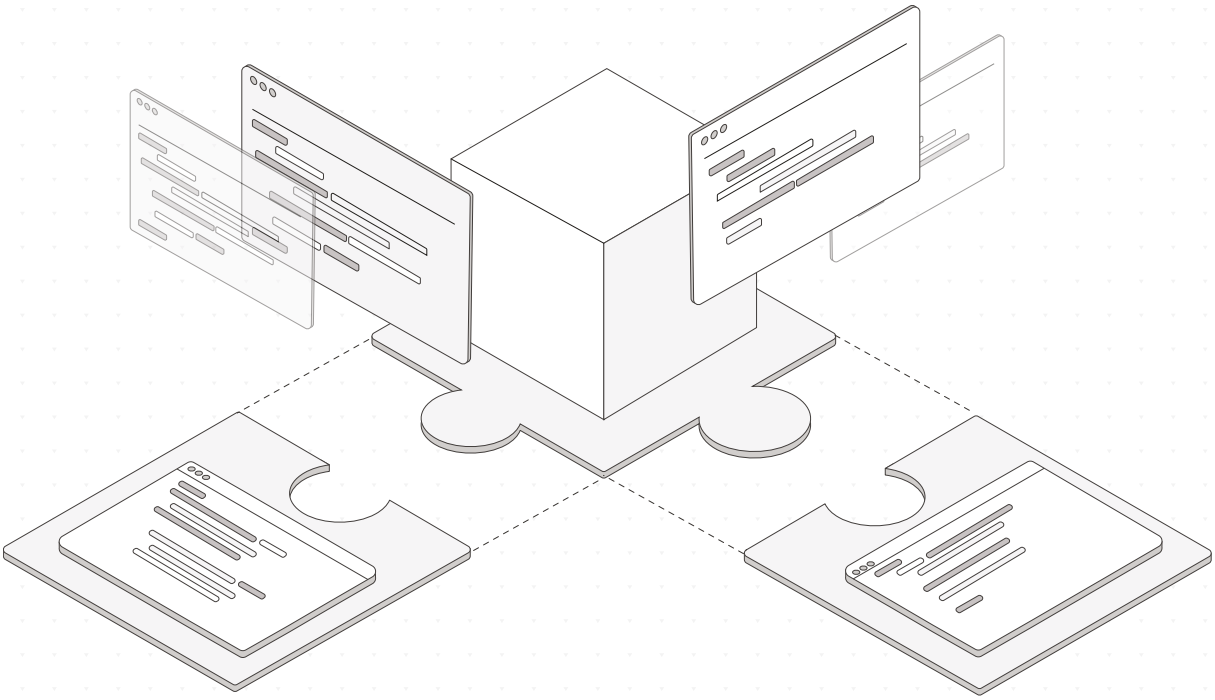
We'll explore three common scenarios: a plain JavaScript application with no native dependencies, a TypeScript application that requires a build step, and an application that includes native addons, which require compilation during installation.

7.2 TypeScript Application

For TypeScript applications, a multi-stage build is recommended. In the first stage, development dependencies are installed to compile the source code.

In the final production stage, only the compiled output and required runtime dependencies are copied, resulting in a smaller, more secure image

```
# The build image

FROM node:22.10.0 AS build
RUN apt-get update && apt-get install -y --no-install-recommends
dumb-init

WORKDIR /usr/src/app
ENV NODE_ENV=production

COPY package.json .
COPY package-lock.json .
COPY .npmrc .
COPY tsconfig.json .

RUN npm install

COPY src ./src

RUN npm run build
RUN npm prune --production

# The production image

FROM node:22.10.0-slim

COPY --from=build /usr/bin/dumb-init /usr/bin/dumb-init

USER node
WORKDIR /usr/src/app

COPY --chown=node:node --from=build /usr/src/app/node_modules /usr/
src/app/node_modules
COPY --chown=node:node --from=build /usr/src/app/dist /usr/src/app

ENV NODE_ENV=production

ENV APP_HTTP_PORT=3000
EXPOSE ${APP_HTTP_PORT}

CMD [“dumb-init”, “node”, “app.js”]
```



7.3 Application with native addons

For Node.js applications that rely on native addons, such as image processing libraries like `sharp` or `prisma`, it is recommended to use a multi-stage Docker build. In this approach, the build stage includes all development dependencies required to compile native modules. The final production stage includes only the compiled output and the necessary runtime dependencies, resulting in a smaller and more secure image.

Some libraries, such as `sharp`, provide precompiled binaries that automatically match the deployment platform. Others, like `prisma`, compile native components during installation, which requires that the build environment closely match the production environment to avoid runtime incompatibilities.

Base Image Considerations

There are two main base image options for applications with native addons:

`node:alpine` (lightweight): Based on Alpine Linux and using the musl C standard library, this image offers a small footprint. Prisma supports Alpine by downloading musl-compatible engines. However, it is important not to install glibc (e.g., via the `libc6-compat package`) in this image, as doing so can cause some libraries such as `prisma` to malfunction.

`node:slim` (stable): Based on Debian and using the glibc standard library, this image offers broader compatibility with native modules. It is a stable and widely supported choice. Some older versions may require the manual installation of system libraries, such as `libssl`.

A common best practice is to use the full `node:slim` image during the build stage and switch to a smaller compatible image, either `node:slim` or `node:alpine`, depending on dependency support, for the runtime stage.

```
# The build image

FROM node:22.10.0 AS build
RUN apt-get update && apt-get install -y --no-install-recommends
dumb-init openssl

WORKDIR /usr/src/app

COPY package.json .
COPY package-lock.json .
COPY .npmrc .

RUN npm install

COPY src ./src

RUN npm prune --production

# The production image

FROM node:22.10.0-slim

COPY --from=build /usr/bin/dumb-init /usr/bin/dumb-init

USER node
WORKDIR /usr/src/app

COPY --chown=node:node --from=build /usr/src/app/src /usr/src/app
COPY --chown=node:node --from=build /usr/src/app/node_modules /usr/
src/app/node_modules

ENV NODE_ENV=production

ENV APP_HTTP_PORT=3000
EXPOSE ${APP_HTTP_PORT}

CMD [“dumb-init”, “node”, “app.js”]
```

7.4 Pushing the Docker Image to a Container Registry

Once the Docker image is built, it needs to be tagged and pushed to a container registry. From a container registry, the image can be deployed. Commonly used container registries include: Docker Hub, Amazon Elastic Container Registry (ECR), Google Container Registry (GCR), and Github Container Registry (GHCR).

```
# Build the Docker image
docker build -t my-dockerhub-username/my-node-app:latest .
# Push the image to Docker Hub
docker push my-dockerhub-username/my-node-app:latest
```

7.5 Choosing the Right Cloud Deployment Model for Node.js

Choosing the right cloud deployment model depends on scalability needs, cost considerations, and operational complexity.

Platform as a Service (PaaS)

PaaS solutions abstract away infrastructure management, allowing developers to focus on writing code. Popular PaaS options for Node.js applications include:

Google Cloud Run
Runs stateless containers and scales automatically.

AWS Lambda (when used with API Gateway)
Executes Node.js functions in a fully managed, serverless environment.

Azure Functions
Provides event-driven, serverless execution for Node.js applications.

Red Hat OpenShift
A Kubernetes opinionated configuration with hybrid and multi-cloud support

Benefits of PaaS:

- Simplifies deployment and reduces operational overhead.
- Built-in scaling and load balancing.
- No need to manage underlying servers.

Challenges of PaaS:

- Opinionated and lacking flexibility.
- Higher monetary cost.

Infrastructure as a Service (IaaS)

IaaS solutions provide virtual machines (VMs) and networking resources, allowing developers to deploy Node.js applications with full control over the environment. Common IaaS providers include:

Amazon EC2 (AWS)

Google Compute Engine (GCP)

Azure Virtual Machines (Azure)

Benefits of IaaS:

- Greater flexibility in configuring the server environment.
- Full control over security policies and networking.
- Suitable for applications with specific performance or compliance requirements.

Challenges of IaaS:

- Requires manual provisioning, configuration, and scaling.
- Increased operational complexity compared to PaaS solutions.

Serverless Functions

Serverless functions are a type of PaaS which have a narrow application scope. They are best suited for workloads that run long when there is an active user or workloads that can be highly concurrent and are only required in short bursts.

Benefits of Serverless Functions:

- Guide developers to simple, composable software.
- Low-use workloads can be operated cheaply.

Challenges of Serverless Functions:

- If an application isn't receiving constant traffic, every request can trigger a cold start, adding noticeable latency.
- Not suited for complex applications, requiring a significant amount of infrastructure wiring to work correctly.
- While serverless is pay-per-use, this pricing model quickly becomes more costly than running a simple, auto-scaling server once you have any significant traffic.

Container Orchestration

Kubernetes is the most well-known orchestration system, there are a number of others that perform similar functions. With container orchestration, you get a cross between IaaS and PaaS. Common providers include:

Google Kubernetes Engine (GCP)

Amazon Elastic Kubernetes Service (AWS)

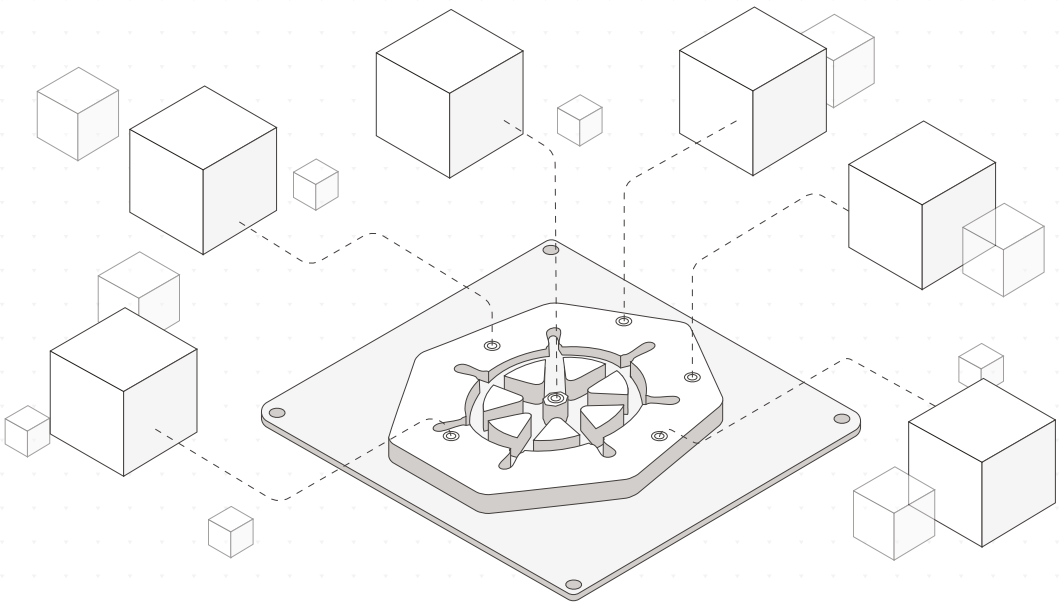
Nomad (HashiCorp)

Benefits of Container Orchestration:

- Reduced operational overhead compared to IaaS.
- Increased flexibility compared to PaaS.
- Standardised resource management.
- Vast ecosystem of tools.

Challenges of Container Orchestration:

- Steep learning curve due to abstraction of resources and applications.
- Every provider uses slightly different terminology.



7.6 Kubernetes and Container Orchestration 101

Kubernetes (K8s) is an open-source container orchestration platform that automates deployment, scaling, and management of containerized applications. It is the most popular container orchestration platform on the market. Kubernetes is, for the most part, the same across all installations but can differ in how storage and networking are supplied by the cloud provider.

Containers can be thought of as the definition of the Operating System and application being run. In Kubernetes, containers are run on Pods, which act like Virtual Machines. Pods provide the compute, storage, and network resources for the containers. The Pod interface provides a way for Kubernetes to create replicated sets and dynamically scale vertically.

7.7 Deploying Node.js in Kubernetes

Kubernetes has become the standard for large teams as well as teams with large portfolios.

When using Kubernetes, the following features are gained:

Automated Scaling
Adjusts the number of running containers based on application relevant metrics.

Self-healing
Restarts failed containers and replaces unhealthy pods.
Load balancing: Distributes traffic across multiple instances to prevent bottlenecks.

Service Discovery
Simplifies networking between services using internal DNS and a well-defined naming scheme.

Safe deployments
Deployment rollout is stopped if a pod is found to be failing, avoiding broken applications getting into production.

Requirements

1. A hosted image that can be downloaded by Kubernetes. Docker Hub provides an easy to use container registry.

2. A Kubernetes installation. For local development, `k3d` is an excellent tool. A working cluster can be started with `k3d cluster create demo -p "30303:30303"`

2. The `kubectl` application

Deploying

Deploying an application consists of several Kubernetes resources:

- One or more `Pod`
- A `ReplicaSet`
- A `Deployment`
- A `Service`

At the core, an application runs inside a container that is hosted by a `Pod`. A `Pod` is a single instance and can be thought of as a Virtual Machine. A single instance of an application is not helpful for scalability and resilience though.

To run multiple instances, a `ReplicaSet` is used. This resource controls the number of running Pod by using a template to define the `Pod` and making sure a set number of replicas are always available.

Setup

It is a good idea to use a new namespace for experimentation.

This can be done through `kubectl` directly or by applying a resource YAML.

```
kubectl create namespace cloud-nodejs-demo
```

A simple YAML file looks like this:

```
apiVersion: v1
kind: Namespace
metadata:
  name: cloud-nodejs-demo
```

Save this file as namespace.yaml and apply to the cluster:

```
kubectl apply -f namespace.yaml
```

Deployment

The next step is to get an application up and running. The best practice for replicated instances is to use a `Deployment` resource. This manages a `ReplicaSet` and allows for simple update and removal strategies of `Pod` resources.

Create a file called deployment.yaml with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-app
  namespace: cloud-nodejs-demo
  labels:
    app.kubernetes.io/instance: demo-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app.kubernetes.io/instance: demo-app
  template:
    metadata:
      labels:
        app.kubernetes.io/instance: demo-app
    spec:
      containers:
        - image: “docker.io/platformatic/cloud-nodejs-demo:latest”
          name: demo-app
          ports:
            - containerPort: 3000 # Port the app is listening on
```

Make sure to replace the `image` with another name. Create the resource:

```
kubectl apply -f deployment.yaml
```

Make sure to replace the `image` with another name. Create the resource:

```
> kubectl --namespace=cloud-nodejs-demo get pods
NAME                          READY   STATUS    RESTARTS   AGE
demo-app-77b7bb9cdc-flf25     1/1     Running   0          26s
demo-app-77b7bb9cdc-knbqp     1/1     Running   0          26s
demo-app-77b7bb9cdc-w5fgz     1/1     Running   0          26s
```

Access

Network access to the deployed application can become complicated very quickly. The idea is to create a `Service` resource which acts as an in-cluster router to the deployed pods. The simplest method is to use a `NodePort` which allows for defining a port to access the application.

This isn't a good long-term solution but does provide a quick way to verify the application works as expected. A `Running` application is not necessarily a working application.

Create a file called service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: cloud-nodejs-demo
  namespace: cloud-nodejs-demo
  labels:
    app.kubernetes.io/instance: demo-app
spec:
  type: NodePort
  selector:
    app.kubernetes.io/instance: demo-app
  ports:
    - port: 3000
      nodePort: 30303
```

Create the resource:

```
kubectl apply -f service.yaml
```

The application access can be verified with a curl request:

```
> curl "http://127.0.0.1:30303"
Hello, World!
```

A `NodePort` service should only be a temporary measure. Production-level services typically run as `LoadBalancer` type.

7.8 Exposing an application

- Cover local development (/etc/hosts, load balancer, ingress)
- Differences with AWS, GCP, and Azure

Each cloud provider has a different way of providing public internet access to a Deployment. All major providers support the Ingress resource though each cloud provider will have slight variations in the annotations used.

For GCP:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: demo-app
  annotations:
    kubernetes.io/ingress.class: gce
spec:
 rules:
  - host: demo-app.example
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: demo-app
            port:
              number: 3000
```

For AWS:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: demo-app
  annotations:
   alb.ingress.kubernetes.io/scheme: internet-facing
   alb.ingress.kubernetes.io/target-type: ip
spec:
 ingressClassName: alb
 rules:
  - host: demo-app.example
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: demo-app
            port:
              number: 3000
```

For Azure:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: demo-app
  annotations:
   kubernetes.io/ingress.class: azure/application-gateway
spec:
 rules:
  - host: demo-app.example
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: demo-app
            port:
              number: 3000
```

Next level

To get Kubernetes deployments ready for production, a good first step is turning all resource creation into templates using Helm.

Using the `Ingress` as an example, the template might look like:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ $.Values.appName }}
  annotations:
   {{ if eq “gcp” $.Values.cloud }}
    kubernetes.io/ingress.class: gce
   {{- end }}

   {{ if eq “aws” $.Values.cloud }}
   alb.ingress.kubernetes.io/scheme: internet-facing
   alb.ingress.kubernetes.io/target-type: ip
   {{- end }}

   {{ if eq “azure” $.Values.cloud }}
   kubernetes.io/ingress.class: azure/application-gateway
   {{- end }}
  labels:
    app.kubernetes.io/instance: {{ $.Values.appName }}
spec:
 rules:
  - host: demo-app.example
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: {{ $.Values.appName }}
            port:
              number: 3000
```

With different `values.yaml` files, any number of applications can be deployed with a standardized configuration. A values file for this example would look like:

```
appName: demo-app
cloud: gcp
```

7.9 The Fundamental Mismatch Between Kubernetes and Node.js

Kubernetes is designed as an infrastructure-first platform—it scales applications based on CPU and memory usage. However, Node.js applications don't operate like traditional CPU-bound processes.

The event-driven model of Node.js allows it to handle thousands of concurrent requests on a single thread, making CPU usage an unreliable scaling metric.

Key Challenges:

Inefficient Scaling
CPU-based scaling can misinterpret Node.js workloads, triggering unnecessary pod creations.

Memory Management Conflicts
Kubernetes enforces strict memory limits, while Node.js' V8 engine dynamically manages heap memory, leading to premature pod termination.

CPU vs Event Loop Utilization

CPU utilization is a common way to measure Kubernetes applications. This makes sense as CPU utilization is built into Kubernetes and is very fast to process.

For Node.js applications, CPU utilization is not a useful metric because it does not directly correlate with application load. The CPU can spike from application load, garbage collection, or async tasks. By measuring only CPU for Node.js applications, Kubernetes leans towards over provisioning resources.

A better metric to measure is the Event Loop Utilization (ELU). This metric provides a direct view of how the event loop is operating. A high utilization, sustained for more than a second, can indicate too heavy of a load and an increase in latency.

Measuring ELU requires the correct metrics to be exported and Kubernetes Custom Metrics to be enabled. Using a tool like Platformatic's Intelligent Autoscaler, can simplify the process of setting up scaling while also using industry best practices.

Memory Management

Node.js relies on the V8 engine's garbage collector, which dynamically expands and retains memory based on demand. Unlike traditional applications, V8 does not immediately release memory after use—it only does so when absolutely necessary. This behavior often leads to confusion, as a sudden spike in memory usage does not necessarily indicate a memory leak.

A common mistake many organizations make is killing Node.js processes when memory usage spikes, assuming the application is misbehaving. However, this can lead to unnecessary downtime and instability. The key takeaway is that high memory usage in Node.js is not inherently bad—it's often just a reflection of how V8 optimizes performance by retaining memory.

Memory management for Node.js in Kubernetes requires a long-term strategy. Avoid setting aggressive memory limits at first. Start by monitoring heap usage instead of total memory consumption. With enough data, it is possible to tune the memory limits and optimize garbage collection.

7.10 Avoiding Kubernetes Pitfalls with Node.js

1. DNS Resolution Overload

Kubernetes does not cache DNS lookups by default, meaning every HTTP request between microservices triggers a fresh DNS resolution.

This can overwhelm the internal DNS resolver.

Fix: Implement a local DNS cache on Kubernetes nodes to reduce unnecessary lookups and improve response times.

7.11 Serverless Functions with Node.js

Serverless computing allows developers to execute code in response to events without managing servers. Cloud providers handle provisioning, scaling, and maintenance.

Advantages of Serverless Node.js

Pay-per-use
Charges are based on execution time, reducing costs for infrequent workloads.

Automatic scaling
Functions scale up and down based on demand.

Reduced operational overhead
No need to manage infrastructure.

Limitations of Serverless Computing

Cold starts
Initial function invocation can be slow due to resource allocation.

Execution time limits
Most providers impose a maximum runtime (e.g., AWS Lambda limits executions to 15 minutes).

Limited execution environment
Serverless functions may not support complex configurations or long-running processes.

At near-zero scale, serverless might seem like an attractive option because you only pay when traffic happens. However, the reality is that if you're **at that scale, you're also experiencing cold starts every time**, making performance significantly worse. In contrast, even a cheap, auto-scaled EC2 instance can provide better performance and cost-effectiveness for most applications. Many bootstrapped startups skip serverless entirely, opting for lightweight instances that offer more control and efficiency with negligible cost differences.

The best approach is to build in a way that makes migration easy. Using something like `fastify-aws-lambda` ensures that your code can be moved out of Lambda into a normal server-based environment without major rewrites.

Starting with serverless can be fine if it encourages modularity, but as soon as your application grows, you'll likely want to migrate to something more efficient.

The following is an example of an application running on AWS Lambda with the `@fastify/aws-lambda` plugin

`app.js`

```
import { randomUUID } from 'node:crypto'
import fastify from 'fastify'
import { DynamoDBClient } from '@aws-sdk/client-dynamodb'
import {
  DynamoDBDocumentClient, GetCommand, PutCommand,
  UpdateCommand, DeleteCommand
} from '@aws-sdk/lib-dynamodb'

const PRODUCTS_TABLE = process.env.PRODUCTS_TABLE

function build(options = {}) {
  const app = fastify(options)

  const client = new DynamoDBClient({
    endpoint: process.env.DYNAMODB_ENDPOINT,
    region: process.env.AWS_REGION || 'us-east-1'
  })

  const dynamoClient = DynamoDBDocumentClient.from(client)

  // GET /products/:id - Get a specific product
  app.get('/products/:id', async (request, reply) => {
    try {
      const command = new GetCommand({
        TableName: PRODUCTS_TABLE,
        Key: { id: request.params.id }
      })

      const response = await dynamoClient.send(command)
      const product = response.Item

      if (!product) {
        reply.code(404)
        return { message: 'Product not found' }
      }

      return product
```

```
    } catch (err) {
      request.log.error({ err }, 'Error retrieving product')
      reply.code(500)
      return { message: 'Error retrieving product' }
    }
  })

  // POST /products - Create a new product
  app.post('/products', async (request, reply) => {
    const product = validateProduct(request.body)

    try {
      const command = new PutCommand({
        TableName: PRODUCTS_TABLE,
        Item: { id: randomUUID(), ...product }
      })

      await dynamoClient.send(command)
      reply.code(201)
      return product
    } catch (err) {
      request.log.error({ err }, 'Error creating product')
      reply.code(500)
      return { message: 'Error creating product' }
    }
  })

  // DELETE /products/:id - Delete a product
  app.delete('/products/:id', async (request, reply) => {
    try {
      const deleteCommand = new DeleteCommand({
        TableName: PRODUCTS_TABLE,
        Key: { id: request.params.id }
      })

      await dynamoClient.send(deleteCommand)
      return { message: 'Product deleted successfully' }
    } catch (error) {
      request.log.error(error)
      reply.code(500)
      return { message: 'Error deleting product' }
    }
  })

  return app
}

export default build
```

lambda.js

```
import awsLambdaFastify from 'aws-lambda-fastify'
import app from './app.js'

const server = app()
const proxy = awsLambdaFastify(server)

export const handler = async (event, context) => {
  return proxy(event, context)
}
```

Wrapping Up

Running Node.js applications in the cloud provides flexibility, scalability, and cost-effectiveness. Choosing the right deployment strategy depends on the specific requirements of a project:

- **PaaS** is ideal for rapid development and automatic scaling without infrastructure management.
- **IaaS** provides full control but requires more configuration and maintenance.
- **Serverless computing** is best for event-driven applications with unpredictable traffic.
- **Kubernetes** is a powerful solution for large-scale, containerized applications requiring advanced orchestration but comes with added complexity.

By selecting the right cloud deployment model, developers can optimise performance, cost, and operational efficiency for their Node.js applications.

How can we help?

Platformatic empowers teams to innovate by simplifying Node.js operations and accelerating time-to-market.

Our platform provides a comprehensive solution for managing Node.js applications on Kubernetes, enabling teams to:

Reduce development time

By automating routine tasks and providing a unified platform, Platformatic frees developers to focus on building features and delivering value.

Improve application performance

Our Intelligent Autoscaler and optimization tools ensure that your Node.js applications perform at their peak.

Enhance developer experience

Platformatic's user-friendly interface and seamless integration with popular frameworks make it easy for developers to work with.

08

Ensuring Scalability and Resilience within Node.js Applications

Architectural considerations for delivering exceptional user experiences

08

Ensuring Scalability and Resilience within Node.js Applications

Architectural considerations for delivering exceptional user experiences

Nowadays, digital product user expectations are higher than ever, and online services are expected to be available around the clock. This renders scalability and resilience not just desirable qualities but essential requirements for successful Node.js applications.

Scalability refers to the ability of an application to handle increasing workload and user demand without sacrificing performance, while resilience pertains to the application's ability to withstand and recover from failures or disruptions in its environment.

The importance of scalability and resilience cannot be overstated in the context of Node.js applications, particularly in high-traffic environments where sudden spikes in user activity or unforeseen events can significantly impact application performance and availability.

Consider, for example, a popular e-commerce website experiencing a surge in traffic during a holiday sale, or a social media platform facing increased usage during a viral event. In such scenarios, a lack of scalability can lead to sluggish performance, downtime, and ultimately, a loss of revenue and customer trust.

Similarly, resilience is equally crucial for Node.js applications, as even the most meticulously designed systems are susceptible to failures, whether due to hardware malfunctions, software bugs, or network issues. An application's ability to gracefully handle and recover from such failures is essential for maintaining service availability and ensuring a seamless user experience.

This chapter looks at the core concepts and best practices for architecting and developing robust, scalable Node.js applications that can withstand high traffic volumes and unexpected events.

We'll explore various architectural considerations that foster scalability and resilience. By following the guidance in this chapter, you'll equip yourself with the knowledge and tools to build enterprise-grade Node.js applications that can handle ever-growing demands while delivering exceptional user experiences.

8.1 Architectural Considerations for Building Scalable and Resilient Node.js Applications

Microservices Architecture

Microservices architecture has gained popularity in the context of Node.js applications due to its ability to address the limitations of monolithic architectures and provide greater scalability and resilience. In this approach, monolithic Node.js applications are decomposed into smaller, independently deployable services, with each service responsible for a specific functionality or business domain.

By breaking down a Node.js application into smaller, more manageable components, each focussing on a specific functionality- microservices- this enables teams to iterate and scale individual services independently.

This promotes agility and flexibility, allowing Node.js applications to adapt to changing requirements and scale components as needed without affecting the entire system's performance.

Asynchronous Programming:

Node.js's event-driven, non-blocking I/O model is a fundamental feature that enables efficient handling of concurrent requests. By leveraging asynchronous programming techniques such as callbacks, promises, and async/await, Node.js applications can perform non-blocking operations, allowing them to handle high concurrency and improve responsiveness under heavy loads.

Asynchronous programming minimizes the time spent waiting for I/O operations to complete, maximizing resource utilization and improving overall application performance.

This approach is particularly beneficial for building scalable and resilient Node.js applications that can handle large volumes of concurrent requests without becoming overwhelmed.

```
import { writeFileSync } from 'fs'
import { writeFile } from 'fs/promises'

// The first write will complete before the second begins.
writeFileSync('first.file', 'foo')
writeFileSync('second.file', 'bar')

// Both writes will begin immediately but he app will be allowed to
// continue operating on other things while the writes are processed
// to completion in the background.
await Promise.all([
  writeFile('first.file', 'foo'),
  writeFile('second.file', 'bar')
])
```

8.2 Enabling Adaptability: Scaling to Meet Demand

As traffic patterns change, so should your deployment. When load is high, you don't want your application to collapse under the stress. When load is low, you don't want to be paying for more resources than you need, leading to a massively inflated cloud bill.

You need to be prepared for horizontal scaling–that is, adding more servers to the cluster running your application.

At times, you may want to reach for vertical scaling, deploying larger instances when you need more headroom for average workloads, but typically, horizontal scaling is what you'll be reaching for more often and even automatically as adding more instances doesn't require working around inflight traffic.

Horizontal Scaling:

Horizontal scaling involves distributing application load across multiple instances or servers to accommodate increased traffic and workload. By deploying multiple instances of the application and implementing load balancers to evenly distribute incoming requests, organizations can achieve seamless expansion and improved fault tolerance.

Horizontal scaling allows applications to scale out horizontally as demand increases, ensuring uninterrupted service availability and improved performance. This approach enables organizations to handle spikes in traffic and accommodate growing user bases without experiencing degradation in performance or downtime.

For example, scaling a Node.js application in Kubernetes with a HorizontalPodAutoscaler would look something like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nodejs-app
  template:
    metadata:
      labels:
        app: nodejs-app
    spec:
      containers:
      - name: nodejs-app
        image: your-dockerhub-username/nodejs-app:latest
        ports:
        - containerPort: 3000
        resources:
          requests:
            cpu: “200m”
            memory: “256Mi”
          limits:
            cpu: “500m”
            memory: “512Mi”
        env:
        - name: NODE_ENV
          value: “production”
---
apiVersion: v1
kind: Service
metadata:
  name: nodejs-app-service
spec:
  selector:
    app: nodejs-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: LoadBalancer
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nodejs-app-hpa
spec:
  scaleTargetRef:
```

```
    apiVersion: apps/v1
    kind: Deployment
    name: nodejs-app
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

While that can get you a good amount of the way to effective scalable workloads, it's important to understand that the Kubernetes HorizontalPodAutoscaler only operates on high-level system metrics like cpu and memory usage.

It lacks awareness of event loop utilization which is a more accurate measurement of hardware resource utilization.

Vertical Scaling/Sizing:

Vertical scaling involves increasing the capacity of a single server or instance by upgrading its CPU, RAM, disk storage, or network bandwidth to handle greater workloads. Instead of distributing traffic across multiple instances, vertical scaling enhances the power of an individual machine to process more requests efficiently.

By adding more resources to a single Node.js application instance, organizations can improve performance, reduce latency, and simplify infrastructure management. Vertical scaling is particularly useful for applications with stateful dependencies, monolithic architectures, or workloads that require high-speed data processing on a single node.

However, vertical scaling has limitations—it's bounded by the maximum hardware capacity of the machine and may result in higher costs as resource upgrades become increasingly expensive.

Additionally, scaling up a single instance introduces a single point of failure, making redundancy and failover mechanisms essential for maintaining availability.

Deploying with auto-scalers

Whether it be with Kubernetes or with a cloud vendor, you should always prepare your deployment systems with quick, painless, and ideally automated scaling in mind. Just checking the box of auto-scaling on its own is not enough, as not all auto-scalers are created equal. You want to understand what is the time-to-life for the environment you target.

Is it a functions-as-a-service environment with cold starts? How long do those cold starts take? How often do you encounter them? How many instances of those functions can be started at any given time (every hyperscaler has a limit)?

Is it a container orchestrator? How long does it take to schedule the workload and have the service come online?

When reacting to scale you need to be sure you can react fast enough. If you have a fast spike of traffic, you may not have time to scale up. In this situation, you may have to plan a scale up ahead of an expected spike, or, if your spikes are less predictable, you may need to over-provision enough to provide a buffer to take enough of the initial climb to compensate for the spin up time for more instances.

Clouds and Kubernetes provide a best-effort scaling strategy based on system-level metrics, but that often lacks the full picture of how a Node.js application is actually behaving as the event loop plays a critical role in the true performance of a Node.js process.

At Platformatic, we built autoscaling into the Command Center to apply scaling decisions to a Kubernetes cluster from a much deeper understanding of the performance characteristics of your Node.js applications.

Your load, in balance

Load balancers are an essential component of horizontal scaling. If you are running in the cloud, your provider likely already manages this. However, if you're deploying on your own hardware, you will need to make sure your traffic can be balanced effectively over as many application instances as you have available.

If you're running on Kubernetes, you're likely going to rely on your Ingress for load balancing, but there may be cases where you want to use per-service load balancers.

Traefik is a popular option for providing more control and visibility than the default Ingress Controller. If you're running on bare metal you may want to consider HAProxy.

Learn about what is available for your target environment and what your load-balancing strategies are. Generally, a simple round-robin strategy that simply rotates between each target instance and directs the single next request is sufficient. However, many load balancers provide more complex strategies which can be more suitable to the characteristics of your traffic profile.

8.3 Monitor the health of your Node.js application with key performance metrics

How will you know it's time to scale if you don't know your application is at risk in the first place? You need to have useful monitoring signals in place to inform you at a glance when the existing scaling strategies you have in place may be insufficient.

You can't scale your way out of a memory leak, you can just slow it down. Similarly, you can't scale your way out of an app that's failing half its requests and magnifying its own traffic with clients applying their own exponential backoff strategies to keep retrying failed requests.

You need to not just know the numbers for your application health, but to understand them too. There's a close relationship between CPU usage and event loop utilization, for example. Understanding how your application behaves under different load patterns is an important step to understanding how to scale it effectively.

To truly understand how your Node.js application is performing and ensure it delivers a seamless user experience, you need to track a specific set of metrics.

So what are these key metrics?

1. Response Time

- Response time refers to the duration between when a request is received by the server and when the corresponding response is sent back to the client. Response time directly impacts user experience. Faster response times lead to more satisfied users, while slower response times can result in frustration and abandonment of the application.

- **Evaluation**: Organizations should aim to monitor response time and set performance targets based on user expectations and industry standards. Continuous monitoring and optimization can help ensure that response times remain within acceptable limits.

2. Throughput

- Throughput measures the number of requests processed by the server within a given time period. Throughput provides insights into the server's capacity to handle incoming requests and its overall efficiency. Higher throughput indicates better scalability and performance.

- **Evaluation:** Organizations should monitor throughput under various load conditions to ensure that the server can handle peak traffic without degradation in performance. Scalability testing can help identify bottlenecks and optimize throughput.

3. Error Rate:

- Error rate refers to the percentage of requests that result in errors or failures. Error rate reflects the stability and reliability of the application. High error rates can indicate bugs, infrastructure issues, or performance bottlenecks that need to be addressed.

- **Evaluation:** Organizations should track error rates over time and investigate spikes or trends to identify underlying issues. Error logging and monitoring tools can help pinpoint the root causes of errors and facilitate troubleshooting.

4. CPU Utilization

- CPU utilization measures the percentage of CPU resources consumed by a Node.js application. High CPU usage can indicate excessive computational overhead, inefficient algorithms, or blocking operations that degrade application performance.

- **Evaluation:** Organizations should track CPU utilization under normal and peak load conditions. Optimizing code, using asynchronous programming, and offloading intensive tasks to worker threads can help reduce CPU consumption and improve application efficiency.

5. RSS Utilization (Resident Set Size)

- RSS (Resident Set Size) represents the amount of memory a Node.js process is actively using in RAM, excluding swapped-out memory. High RSS utilization can indicate memory-intensive processes, inefficient garbage collection, or memory leaks.

- **Evaluation:** Monitoring RSS utilization helps identify memory-related inefficiencies. Organizations should analyze trends, optimize memory usage, and leverage tools like garbage collection tuning and load balancing to prevent excessive memory consumption.

6. Memory spread (heap total vs heap used, old space vs new space)
Memory usage in Node.js is often monitored by tracking the amount of heap memory in use (heap used) compared to the total available heap memory (heap total). The heap is where Node.js stores objects and variables during the execution of an application. Heap total refers to the maximum amount of memory allocated to the heap, while heap used tracks how much of that allocated memory is actively in use.

By monitoring heap total vs. heap used, organizations can identify potential issues such as memory leaks or inefficient memory allocation. If heap used is consistently approaching the heap total limit, it could indicate that the application is consuming excessive memory, which may lead to performance degradation or crashes.

- **Old Space vs. New Space:** Heap memory in Node.js is managed by the V8 JavaScript engine, which divides it into two primary regions:

    1. **New Space:** A smaller, faster allocation area for newly created objects. This space is frequently garbage-collected using a quick "scavenge" process.
    2. **Old Space:** A larger memory region for objects that persist beyond multiple garbage collection cycles. Garbage collection here is slower and more resource-intensive, involving "mark-sweep" and "mark-compact" processes.

Evaluation: Monitoring heap total vs. heap used alongside Old Space vs. New Space helps diagnose memory-related inefficiencies:

- Rapid New Space growth can indicate excessive short-lived object creation, increasing garbage collection frequency and CPU load.
- Uncontrolled Old Space expansion may signal memory leaks or inefficient object retention.

Organizations should leverage profiling tools and memory monitoring tools to optimize memory usage, fine-tune garbage collection, and prevent out-of-memory crashes.

7. Event loop utilization

- Event loop utilization measures the percentage of time the Node.js event loop is actively processing tasks compared to the time it spends idling. The event loop is the heart of Node.js, handling all incoming and outgoing operations. High event loop utilization indicates efficient use of resources and minimal idle time. Conversely, low utilization might suggest inefficiencies, such as waiting for external resources or poorly structured code blocking the event loop.

- **Evaluation:** Organizations should monitor event loop utilization to identify potential bottlenecks that hinder Node.js' asynchronous processing capabilities. Optimizing code for non-blocking operations and leveraging asynchronous libraries can improve event loop utilization and overall performance.

**See Appendix A for a deep dive into the Node.js Event Loop*

8.4 Acting on metrics

When it comes to making decisions based on performance metrics, it's important to ensure that actions such as scaling or terminating instances are done with full consideration of how Node.js behaves. Here are two key principles to guide effective decision-making:

Avoid scaling or killing instances prematurely based on memory consumption

One common mistake is to scale up or terminate an instance when memory consumption reaches a certain threshold, such as 60-80%. However, this can be inefficient in the context of Node.js.

Node.js prioritize runtime performance over memory usage, delaying the "slow" garbage collection (mark&sweep) until it's necessary: the result is that often Node.js will use all the memory allocated to it by the operating system. This means that if more memory is available to the process, Node.js will use it to optimise its performance and handle more operations concurrently.

Node.js doesn't typically "overuse" memory—if the system has available memory, it will utilise it for garbage collection or buffer management, both of which help improve performance. Therefore, killing or scaling an

instance when memory usage is at a moderate level may be premature, especially if the application is still performing well overall. Instead, organisations should focus on whether the memory consumption is truly problematic, such as if it's growing uncontrollably over time (indicating a potential memory leak), or if it's starting to hit the limits of the system's physical memory.

Always consider Event Loop Utilisation (ELU) when scaling based on CPU metrics

Another important consideration is the relationship between CPU usage and application responsiveness. It's common to scale an application based on CPU usage, particularly if CPU usage is at or near 100%. However, high CPU usage in Node.js doesn't necessarily mean the application is unresponsive. Node.js is single-threaded and runs on a non-blocking event loop, which means it can still be responsive even if CPU usage is high, provided the event loop is not blocked.

If the event loop is underutilised, meaning it's not busy processing tasks, scaling the application solely based on CPU metrics could be a misstep. The application may be idle and not in need of additional resources.

Therefore, it's essential to also consider the Event Loop Utilisation (ELU) metric. If the event loop is busy and responding to requests, the system is likely functioning optimally, even if the CPU is at high capacity. Only scale up or take action if ELU reveals that the application is actually experiencing a bottleneck or significant delay in processing.

8.5 Challenges when Monitoring Node.js Performance

Standard monitoring metrics, such as CPU usage and RSS (Resident Set Size), are often insufficient for effective troubleshooting of Node.js applications. These basic metrics provide a high-level view of system performance, but they lack the granularity and context necessary to fully understand the health of a Node.js application, especially in more complex environments.

Monitoring Node.js applications often requires cobbling together insights from multiple tools and custom dashboards, juggling metrics like CPU usage, memory consumption, and latency. This makes it difficult to pinpoint the root cause of slowdowns and unresponsive applications.

Imagine a platform with numerous microservices; a fragmented view might reveal high CPU usage, but without context, identifying the culprit remains a guessing game.

This hinders timely troubleshooting, leading to persistent performance bottlenecks that cost companies millions in lost revenue.

Let's take a deeper look at some of the challenges that arise when monitoring Node.js applications:

Asynchronous Nature

Node.js operates on an asynchronous, event-driven model, which can make traditional monitoring techniques less effective. Monitoring tools must be capable of tracking asynchronous operations and event loops to provide accurate performance insights.

Complex Ecosystem

Node.js applications often rely on a wide range of dependencies, frameworks, and microservices, making it challenging to monitor the entire ecosystem comprehensively.

Coordinating monitoring across various components and services requires careful planning and integration.

Memory Leaks

A memory leak occurs when memory is allocated but not properly released, leading to increased memory usage over time. Unlike a memory usage issue, which can occur due to high traffic or data inflow without necessarily indicating a problem, a memory leak causes steady, uncontrolled growth that can eventually crash the application.

Node.js applications are particularly vulnerable to memory leaks due to their asynchronous, event-driven nature. Since memory allocation and garbage collection can be more complex in asynchronous processes, it can be difficult to determine if increasing memory usage is due to normal traffic or an actual leak.

To distinguish between the two, it's crucial to use metrics like throughput (request volume) and event loop utilisation (ELU). If memory usage grows alongside throughput and the event loop remains responsive, it's likely expected. However, if memory usage increases without corresponding traffic and the event loop is blocked, this could indicate a leak.

8.6 OpenTelemetry Tracing

OpenTelemetry Tracing builds upon the foundation laid by the Dapper paper, a seminal work published by Google in 2010. Dapper introduced the concept of distributed tracing, a revolutionary approach to monitoring performance in complex, microservices-based architectures. Traditional monitoring tools often struggled to track requests that spanned multiple services, making it difficult to pinpoint performance bottlenecks.

Dapper's innovation involved assigning a unique identifier (trace ID) to each request and propagating it throughout the entire request lifecycle. This allowed developers to trace a request's journey across various microservices, identify potential delays or errors at each hop, and gain a holistic understanding of application performance.

The impact of the Dapper paper was significant. It not only provided a practical solution for distributed tracing but also inspired the development of numerous open-source tracing tools like Zipkin and Jaeger. Today, OpenTelemetry builds upon these advancements by establishing a vendor-neutral approach to collecting and exporting telemetry data, including traces initiated by the Dapper tracing model.

Google's active participation in OpenTelemetry development reflects their continued commitment to fostering a healthy and interoperable monitoring ecosystem for modern applications.

Another equally important paper, which laid the groundwork for Dapper, was the 2007 X-Trace paper titled "A Pervasive Network Tracing Framework".

Dapper's paper was the adaptation of this model, made specific to HTTP with the use of HTTP headers as the transport mechanism.

Popular tools:

Zipkin
An open-source distributed tracing tool that helps you trace requests across microservices. Zipkin collects trace data from instrumented applications and backend systems, providing detailed visualisations of request lifecycles. It is especially useful for identifying latency issues and pinpointing performance bottlenecks within your Node.js applications.

Jaeger
A popular distributed tracing solution that offers a scalable, user-friendly interface for analysing trace data. Jaeger collects detailed traces and visualises how requests flow through your application, helping developers identify potential delays, errors, and inefficiencies.

It's highly recommended for applications using microservices architectures, where tracking requests across various components is critical.

8.7 Setting up Monitoring, Tracing, Health with Watt

Watt comes with telemetry out of the box, making it easier to fulfill essential non-functional requirements such as observability and system health. These features are integrated at the framework level, requiring minimal configuration while offering full control when customization is needed.

Monitoring provides insight into how the system behaves over time by collecting and aggregating quantitative data. Tracing adds visibility into the lifecycle of individual requests, highlighting latencies and dependencies across service boundaries.

Health checks ensure that the application signals its status correctly to orchestrators and load balancers, helping to maintain reliability and uptime.

In this section, we will explore how Watt exposes metrics compatible with Prometheus, how it integrates with OpenTelemetry for distributed tracing, and how to enable and customize health endpoints. These tools are essential for building confidence in the behavior of your application in production environments and for proactively detecting anomalies or regressions.

A complete working example demonstrating how to implement monitoring, tracing, and health checks will be provided later in the chapter.

Additional options include setting a custom hostname, enabling authentication, and attaching global labels to all exported metrics. It's also possible to define custom metrics directly in the application code to capture domain-specific insights.

Monitoring

Watt integrates with Prometheus. When enabled, it exposes a `/metrics` endpoint on port `9090` that Prometheus can scrape. The exported metrics include HTTP request durations, request counts, memory usage, event loop lag, and other relevant runtime statistics.

You can also define custom application-specific metrics if needed. No application code is required to start collecting standard metrics. Additional options include setting a custom hostname, enabling authentication, and attaching global labels to all exported metrics. It's also possible to define custom metrics directly in the application code to capture domain-specific insights.

On `watt.json`, you'll only need to set:

```
{
  "$schema": "https://schemas.platformatic.dev/@platformatic/
runtime/2.55.0.json",
  "metrics": true
}
```

Prometheus Configuration

To collect and visualize these metrics, you can run **Prometheus** locally or in Docker. Below is a minimal `prometheus.yml` configuration to scrape metrics from a Watt application:

```
global:
  scrape_interval: 15s
  scrape_timeout: 10s
  evaluation_interval: 1m

scrape_configs:
  - job_name: 'platformatic'
    scrape_interval: 2s
    metrics_path: /metrics
    scheme: http
    basic_auth:
      username: platformatic
      password: mysecret
    static_configs:
      - targets: ['192.168.69.195:9090']
        labels:
          group: 'platformatic'
```

In this configuration:

- The `targets` field should point to the IP and port where Watt exposes metrics.
- Basic authentication is configured to match the Watt telemetry settings.
- The IP address must be reachable from the Prometheus container. If Prometheus runs in Docker on the same host, use the host's LAN IP, not `localhost` or `127.0.0.1`.

Save the configuration to `./prometheus/prometheus.yml`. Then, create a `docker-compose.yml` to run Prometheus:

```
version: "3.7"

services:
  prometheus:
    image: prom/prometheus:latest
    volumes:
      - prometheus_data:/prometheus
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
    ports:
      - '9090:9090'

volumes:
  prometheus_data: {}
```

Run the stack with:

```
docker-compose up -d
```

Then open http://localhost:9090 in your browser. You should see the Prometheus UI, and you can query metrics like:

```
{group="platformatic"}
```

Refer to the Prometheus documentation for details on how to build queries and understand metric types.

Grafana Configuration

To visualize these metrics with **Grafana**, extend your `docker-compose.yml` file to include a `grafana` service:

```
version: “3.7”

services:
  prometheus:
    image: prom/prometheus:latest
    volumes:
      - prometheus_data:/prometheus
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    command:
      - ‘--config.file=/etc/prometheus/prometheus.yml’
    ports:
      - ‘9090:9090’

  grafana:
    image: grafana/grafana:latest
    volumes:
      - grafana_data:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=secret
    depends_on:
      - prometheus
    ports:
      - ‘3000:3000’

volumes:
  prometheus_data: {}
  grafana_data: {
```

Start the full stack:

```
docker-compose up -d
```

Open Grafana at http://localhost:3000, log in with the default user admin and the password you set (`secret`), then:

1. Navigate to **Configuration** → **Data Sources** → **Add data source**
2. Choose **Prometheus**
3. Set the URL to `http://prometheus:9090`
4. Click **Save & Test**

You can now create dashboards and add panels using Prometheus queries. Platformatic metrics will be available and can be used to monitor HTTP performance, CPU, memory usage, and any custom metrics you define in your application.

Tracing

Watt supports OpenTelemetry integration by default, allowing you to export trace data to any OTLP-compatible backend, such as Jaeger, Tempo, or Honeycomb. Tracing helps you understand the flow of requests through your system, identify performance bottlenecks, and gain visibility into distributed interactions.

Watt automatically instruments its HTTP layer, but more specific telemetry instrumentation can be configured depending on the services used within your application. For example, if a service is built using `express`, you can enable the `@opentelemetry/instrumentation-express` package to capture detailed tracing data for that part of the system.

In addition to automatic instrumentation, Watt allows you to add custom tracing for specific operations using the `@opentelemetry/api package`. This is useful when you want to trace internal logic that isn't automatically captured, such as database queries, third-party API calls, or computationally expensive functions.

By manually creating and managing spans, you can enrich your trace data with meaningful, application-specific details.

8.8 A Sample `watt.json` Configuration to Enable Tracing

To enable tracing in Watt and export telemetry data to an OTLP-compatible backend such as Jaeger, you can use the following minimal configuration on `watt.json`:

```
{
  “$schema”: “https://schemas.platformatic.dev/@platformatic/
runtime/2.55.0.json”,
  “telemetry”: {
    “serviceName”: “example”,
    “exporter”: {
      “type”: “otlp”,
      “options”: {
        “url”: “http://localhost:4318/v1/traces”
      }
    }
  }
}
```

This configuration sets up Watt to export trace data to a locally running Jaeger instance using the OTLP HTTP protocol.

Jaeger Setup with Docker Compose

Here’s a basic `docker-compose.yml` file to run **Jaeger** locally with OTLP support enabled:

```
version: ‘3.7’

services:
  jaeger:
    image: jaegertracing/all-in-one:1.48
    container_name: jaeger
    ports:
      - “16686:16686”   # Jaeger UI
      - “4318:4318”     # OTLP HTTP receiver
    networks:
      - watt-network

networks:
  watt-network:
```

Start Jaeger with:

```
docker-compose up -d
```

Verifying Traces

1. Start your Watt application:

```
watt start
```

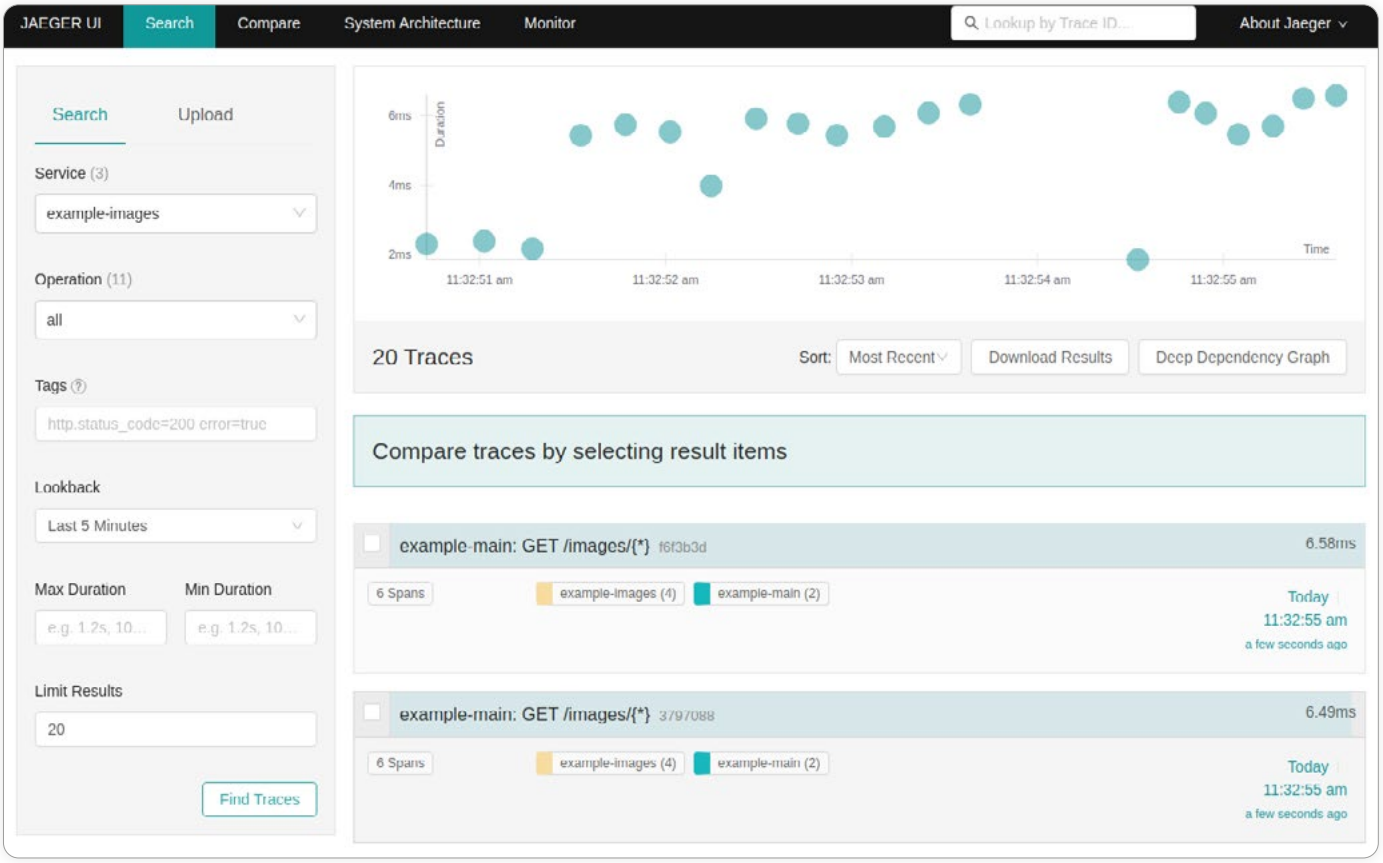
2. Trigger a request, for example:

```
curl http://localhost:3000/images/a-beautiful-sunset
```

3. Open the Jaeger UI in your browser at:

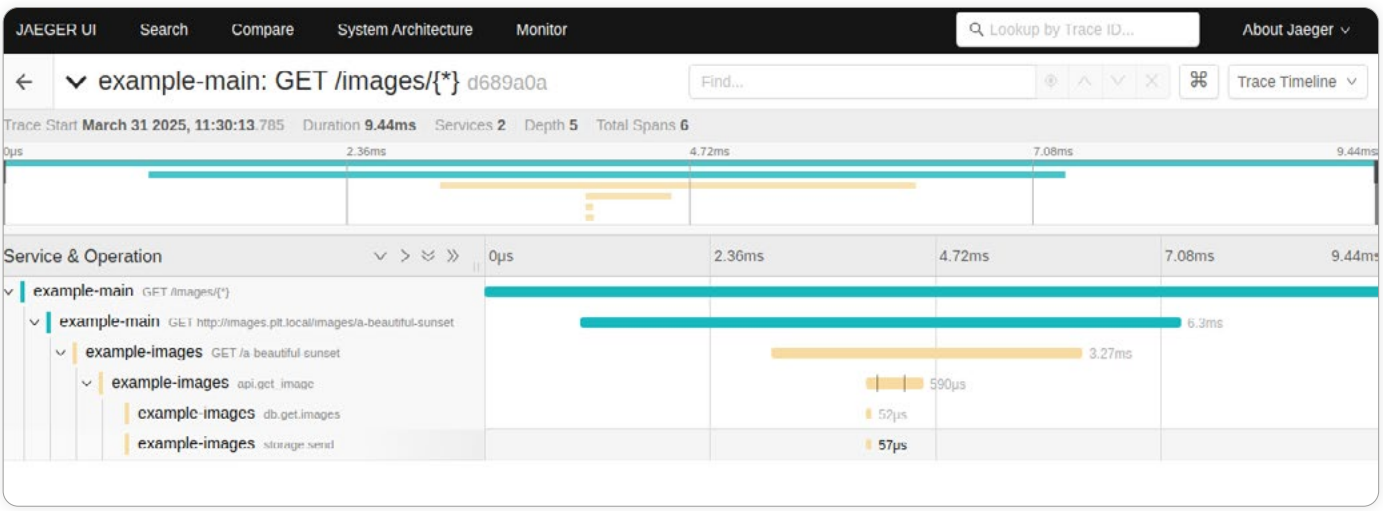
```
http://localhost:16686/
```

Select the service name (example) from the dropdown and search for recent traces. You should see the request you just made, along with the corresponding spans generated by Watt.



The trace of a specific HTTP request can be retrieved, highlighting a particular call.

This facilitates a thorough examination of each phase of the request's lifecycle, from initiation through internal processing to the final response.



Health

Platformatic provides a built-in API for implementing readiness and liveness checks through its metrics server. When telemetry is enabled, the metrics server also exposes two dedicated endpoints for health monitoring:

- `/ready` – indicates whether the service is running and ready to accept traffic.
- `/status` – verifies whether all dependent services in the stack are reachable.

These endpoints are designed to be used by orchestrators such as Kubernetes to monitor the health of the application and manage its lifecycle appropriately. Health checks can also be customized in the application code to meet specific requirements. For example, you can extend the readiness probe to ensure the database is reachable, or verify that an external dependency is responding correctly.

Implementation with Watt

Let's walk through a sample application that includes monitoring, tracing, and health checks.

We'll implement a hypothetical service responsible for fetching images and user profiles. The project follows a modular monolith architecture and uses Platformatic Watt as its runtime environment.

```
app/
├── watt.json
├── services/
    ├── main/                  # Platformatic Composer
    │   ├── platformatic.json
    │   ├── ...
    │   └── package.json
    ├── images/                # Images service
    │   ├── index.js
    │   ├── db.js
    │   ├── storage.js
    │   ├── platformatic.json
    │   └── package.json
    ├── profiles/              # Profiles service
        ├── ...
        └── package.json
```

watt.json

Metrics are exposed on port 9090 at the /metrics endpoint, making them easy to scrape using tools like Prometheus. Tracing data is exported to a Jaeger service via the OTLP protocol.

```
{
  "$schema": "https://schemas.platformatic.dev/@platformatic/
runtime/2.55.0.json",
  "entrypoint": "main",
  "autoload": {
    "path": "services"
  },
  "metrics": {
    "endpoint": "/metrics",
    "port": 9090
  },
  "telemetry": {
    "serviceName": "example",
    "exporter": {
      "type": "otlp",
      "options": {
        "url": "http://localhost:4318/v1/traces"
      }
    }
  }
}
```

The images service is automatically instrumented for telemetry by Watt. However, it also extends its observability features by customizing the health check and adding a custom metric.

- The health check is overridden to verify that both the database and the storage service are operational.
- A custom Prometheus counter is added to track the number of image requests received by the service.

---

services/images/index.js

The images service is automatically instrumented for telemetry, but it can also customize the healthcheck, in this case to ensure the database and the storage is properly working, and also it adds a custom metric to count the images requests

```
import { trace } from '@opentelemetry/api'
import fastify from 'fastify'
import db from './db.js'
import storage from './storage.js'

export function create() {
  const app = fastify()

  // Get tracer for this service
  const tracer = trace.getTracer('images-service')

  let status = true

  globalThis.platformatic.setCustomHealthCheck(async () => {
    status = await Promise.all([
      checkDatabase(),
      checkStorageService()
    ])
    return status
  })

  // Add a counter metric to track the number of images requests
  const { client, registry } = globalThis.platformatic.prometheus
  const counterMetric = new client.Counter({
    name: 'images_requests',
    help: 'Images requests',
    registers: [registry]
  })

  // GET /images/:slug - Get image by slug
  app.get('/:slug', async (request, reply) => {
    return tracer.startActiveSpan('api.get_image', async (span) => {
      try {
        // Set span attributes
        span.setAttribute('http.method', 'GET')
        span.setAttribute('http.route', '/:slug')
        span.setAttribute('http.slug', request.params.slug)

        // Increment Prometheus counter
        counterMetric.inc(1)

        const { slug } = request.params
        span.addEvent('Retrieving image ID from database', { slug })

        const imageId = await db.get('images', slug)
        span.setAttribute('db.image_id', imageId || 'not_found')

        if (!imageId) {
          span.setAttribute('http.status_code', 404)
          span.addEvent('Image not found')
          reply.code(404)
          span.end()
```

```
          return { error: 'Image not found' }
        }

        span.addEvent('Sending image from storage', { imageId })

        // Get image data using the new send method
        const image = await storage.send(imageId)

        // Set appropriate headers and status code
        reply.header('Content-Type', image.contentType)
        reply.code(200)

        span.setAttribute('http.status_code', 200)
        span.setAttribute('http.content_type', image.contentType)
        span.setAttribute('response.size', image.data.length)
        span.end()

        return image.data
      } catch (error) {
        span.recordException(error)
        span.setStatus({ code: trace.SpanStatusCode.ERROR })
        span.end()
        throw error
      }
    })
  })

  return app
}
```

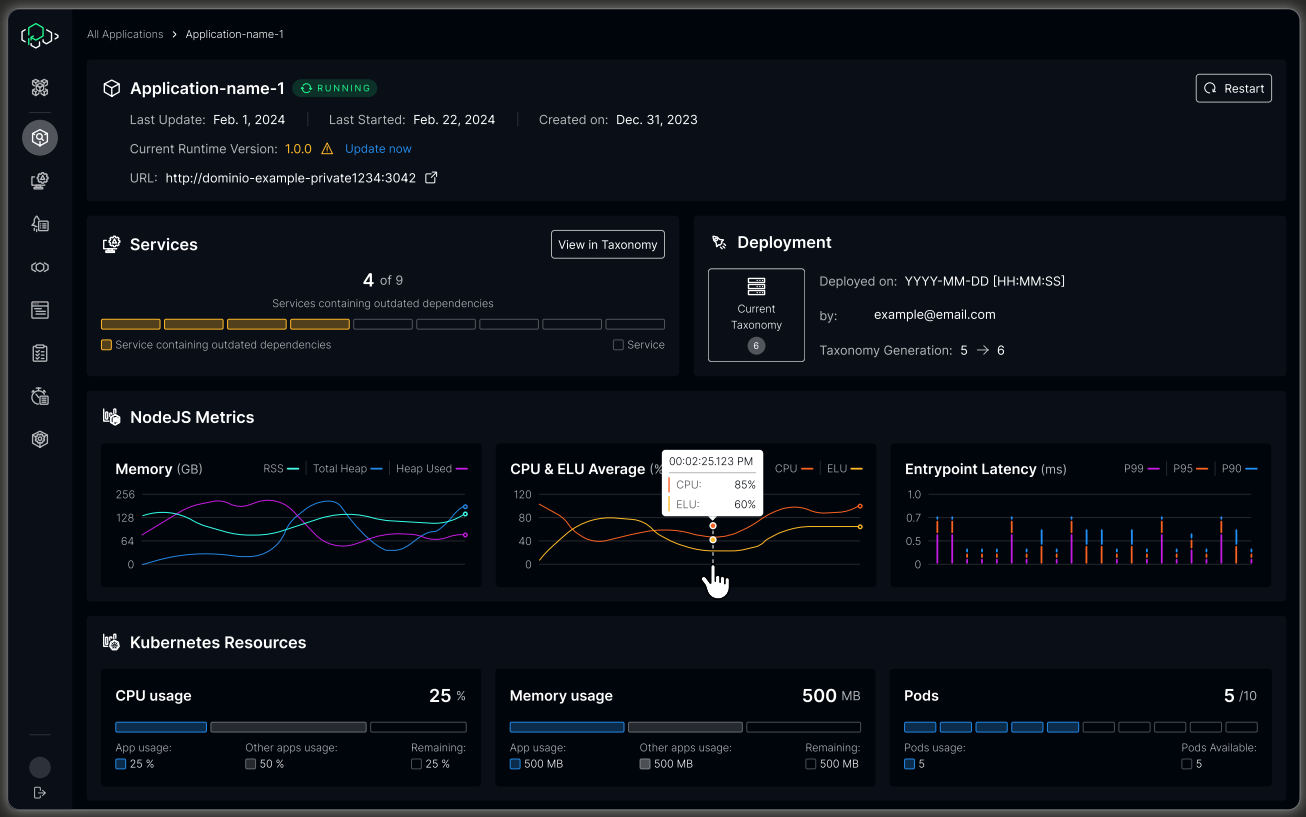
8.9 Commercial Monitoring Products

Numerous commercial monitoring products offer comprehensive solutions for Node.js performance monitoring. These often provide pre-built dashboards, advanced analytics capabilities, and integrations with other development tools. Here are some popular options:

Remember, the best monitoring solution depends on your specific needs and budget. Consider factors like the scale of your application, desired level of detail, and team expertise when making your choice.

How can we help?

The Platformatic Command Center offers a single, intuitive dashboard that gives teams a real-time overview of all their Node.js applications and services so they can see resource utilization, deployment statuses, and key metrics like CPU usage, memory consumption, and latency in one place. By centralizing all application data and activity in one place, the Command Center provides a comprehensive audit trail. This facilitates compliance efforts and helps trace actions for troubleshooting purposes to minimize user impact.



Scale intelligently

Modern web applications experience dynamic traffic surges. Manual resource adjustments for each application are inefficient. The Command Center offers intelligent autoscaling that leverages real-time application data. This goes beyond basic scaling approaches, like those in Kubernetes, which rely on generic cluster-level metrics like CPU or memory usage. While Kubernetes might scale up an entire pod due to high CPU or shut down another due to low memory, this is highly inefficient due to the unique characteristics of Node.js. The Command Center drills down deeper, analyzing Node-specific metrics to pinpoint the exact source of strain. This allows optimal utilization without wasting resources.

8.10 So, how do I build a resilient Node.js app?

Building a resilient Node.js application requires a strategic approach that balances **stability**, **efficiency**, and **adaptability**.

To maintain site uptime and handle traffic spikes effectively, you must anticipate demand fluctuations, optimize resource usage, and enable seamless scaling.

1. Ensuring Stability: A Predictable and Reliable Foundation

Understand Traffic Behavior

Your application's uptime depends on how well you understand and anticipate user behavior. Whether you're an e-commerce platform gearing up for Black Friday or a ride-hailing app preparing for post-event surges, proactive scaling is essential. While market knowledge helps predict trends, real-time monitoring validates assumptions and provides actionable insights.

Proactively Handle Errors

Failures—whether errors, exceptions, or unexpected crashes—should be expected and mitigated. Implement structured logging using tools like Pino, and analyze failure rates to proactively fix vulnerabilities. Logging in a structured format (e.g., JSON) simplifies error tracking, making debugging more efficient.

Optimize Database Access Patterns

Efficient data access prevents performance bottlenecks. Understand whether your workload is read-heavy (e.g., caching precomputed responses) or write-heavy (e.g., logging events). Use tools like Grafana Tempo to analyze slow queries and optimize query performance. Limit and paginate queries where possible to avoid costly operations.

Manage Query Complexity and Indexing

Unbounded query complexity can lead to performance degradation. For GraphQL applications, use Mercurius with query depth-limiting strategies and caching mechanisms. Traditional SQL queries should minimize dynamic JOINs and leverage indexes effectively to avoid full-table scans.

Connection Pooling for Resource Optimization

Opening a new database connection for each request is inefficient and may overwhelm the database. Use connection pooling to manage database load efficiently, balancing performance and resource allocation. This approach also helps mitigate head-of-line blocking, ensuring fair and efficient query execution.

2. Building Efficiency: Optimizing Resource Utilization

Caching at Multiple Levels

A well-designed caching strategy reduces redundant computations and improves response times. Effective caching is layered:

- **CDNs** cache static assets.
- **Query caching** reduces redundant database fetches.
- **Component caching** prevents unnecessary re-renders.
- **Full-page caching** minimizes HTML generation overhead.

For database caching, tools like **async-cache-dedupe** help manage cache expiration and invalidation, while **Mercurius-cache** optimizes GraphQL performance.

Offload Non-Critical Work

Asynchronous task processing prevents long-running operations from blocking main event loop execution. Use message queues for tasks like email notifications or analytics aggregation. This approach improves response times and system resilience, as queued tasks can be retried in case of failures.

Stateless Design for Scalability

Applications holding state in memory face challenges in multi-instance deployments. Store session data in Redis or another key-value store to enable horizontal scaling and prevent load imbalance. Stateless architecture ensures consistency and seamless scaling.

Eliminate Unnecessary Work

Identify tasks that can be removed, optimized, or deferred. For example, replacing session stores with JWTs can eliminate redundant network requests. Strive to design small, focused functions that minimize branching complexity, making performance more predictable and improving code maintainability.

3. Enabling Adaptability: Scaling to Meet Demand

Implement Auto-Scaling

Horizontal scaling—adding more instances—is the preferred approach for handling increased load. Configure auto-scalers in Kubernetes or your cloud environment to dynamically adjust resource allocation. However, not all auto-scalers react quickly enough to traffic spikes; pre-scaling before anticipated surges can prevent outages.

Optimize Load Balancing

Effective load balancing ensures even traffic distribution across instances. In cloud environments, built-in load balancers help manage this, but for self-managed deployments, consider tools like Traefik or HAProxy for greater control. Kubernetes users should optimize Ingress controllers for balanced request routing.

Monitor and Respond to Performance Signals

Scaling decisions should be based on real application health, not just infrastructure metrics. CPU usage, event loop utilization, and request failure rates provide deeper insights into performance bottlenecks. Monitoring tools like Grafana, Prometheus, and Platformatic's Command Center offer real-time observability for smarter scaling strategies.

Wrapping Up

A robust Node.js application requires a balance of stability, efficiency, and adaptability: Understand your traffic patterns to prepare for spikes and lulls.

- Ensure stability with proactive error handling and database optimizations.
- Optimize for efficiency by leveraging caching, connection pooling, and asynchronous processing.
- Enable adaptability through auto-scaling, load balancing, and real-time monitoring.

By following these best practices, you can minimize downtime, optimize performance, and build a scalable, resilient Node.js application that meets the demands of modern web traffic.

09

Using Platformatic to Solve Node for Enterprise

Let's build better Node.js applicatioans — together.

09

Using Platformatic to Solve Node for Enterprise

Let's build better Node.js applications — together.

As we've explored throughout this ebook, Node.js is a robust foundation for building fast, modern applications — but when you're operating in a large-scale, high-stakes environment, things get complicated quickly.

Managing large codebases, optimizing event loop behavior, maintaining preview environments, ensuring resilience, scaling effectively — all while trying to control cloud costs — demands deep expertise and tooling that understands how Node.js actually works.

9.1 That's why we built Platformatic.

Platformatic is the result of years spent helping teams at Fortune 500s and high-growth companies architect, operate, and scale mission-critical Node.js systems. It's built from our open-source work (Fastify, Pino, Node.js core), hardened in production, and designed specifically to solve the recurring challenges enterprise teams face.

Rather than duct-taping together tools not designed for Node, Platformatic gives you a unified platform that helps you:

Control cloud costs
By autoscaling based on real Node.js signals, like event loop lag, heap usage, not misleading metrics like CPU or memory — helping you avoid both over-provisioning and performance bottlenecks.

Improve observability
With Node.js-specific metrics built-in — no need to set up Prometheus or wrangle OpenTelemetry just to see how your app is doing.

Understand deployment risks
With full API dependency mapping, so you know exactly what will break — and which teams are affected — before hitting “deploy”.

Cache intelligently
with a first-of-its-kind system that uses real-time machine learning to automate caching strategies — eliminating endless meetings, manual config, and costly trial-and-error cycles.

Move faster with confidence
thanks to baked-in best practices, from modular architecture to config management, all optimized for Node.js.

At its core, Platformatic isn’t just another platform — it’s the culmination of lessons learned from building, breaking, and scaling Node.js applications for over a decade.

You’ve already chosen Node.js for its performance and flexibility. Platformatic helps you keep that speed as you scale, without paying the hidden costs of complexity, fragility, or surprise cloud bills.

Let’s build better Node.js applications — together.

I

XA

Appendix A

The Node.js Event Loop

Appendix A

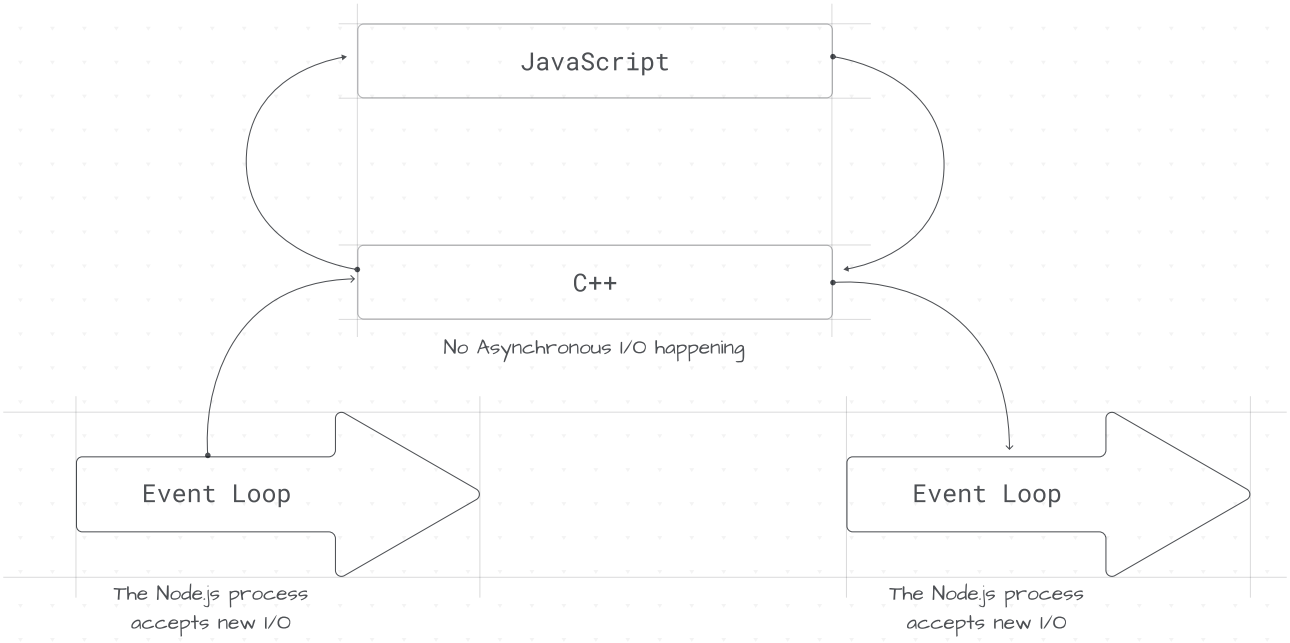
The Node.js Event Loop

The event loop is core to the performance of Node.js, helping it to perform asynchronous and non-blocking operations by leveraging the kernel.

What can go wrong? In this article, we demonstrate a way to stall the event loop and your application as a result.

How could we protect against such a problem? A thorough and accurate understanding of event loops is beneficial for developers to grasp the inner workings of Node.js better.

This article explains the event loop, its importance, and best practices. It also further explains the mathematics behind synchronous response processing and the nitty-gritty of event loop utilization.



A.1 Why is the event loop important in Node.js?

The event loop is important in Node.js for several reasons. First, it forms the pillar of Node's asynchronous architecture, allowing Node to handle multiple concurrent operations without the need for multi-threading efficiently.

Second, the event loop contributes to the performance and resource efficiency of Node.js.

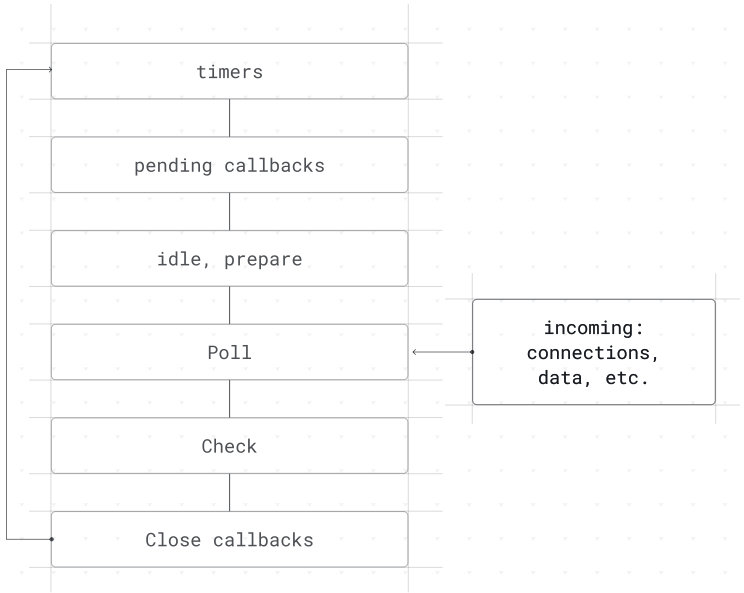
The event loop's non-blocking nature allows developers to write code that can be executed on the available system resources, helping it provide fast responses.

Compared to the thread-based model, the event loop model has a significant advantage: it enables the CPU to handle many more requests at once. It is also more performant than the thread-based model, using a lot less memory to execute for the kernel.

A.2 The Actual Event Loop

The event loop consists of the following phases:

- Timers
- Pending callbacks
- Idle/prepare
- Poll
- Check
- Close callbacks
- Incoming connections and data



The first phase– the timers– are callbacks registered with `'setTimeout()'` or `'setInterval()'`.

They also allow us to monitor the event loop with the option to schedule data, ultimately

offering a good way to check if an event is idle. The event loop then executes expired timers and checks for pending callbacks again.

The I/O callbacks are checked first in the poll phase, followed by the '`setImmediate()`' callbacks and microtasks. Node.js also has a special callback, the `process.nextTick()`, which executes after each loop phase. This callback has the highest priority.

During the poll phase, the event loop looks for events that have completed their asynchronous tasks and are ready to be processed.

We then move to the check phase, during which the event loop executes all the callbacks registered with '`setImmediate()`.

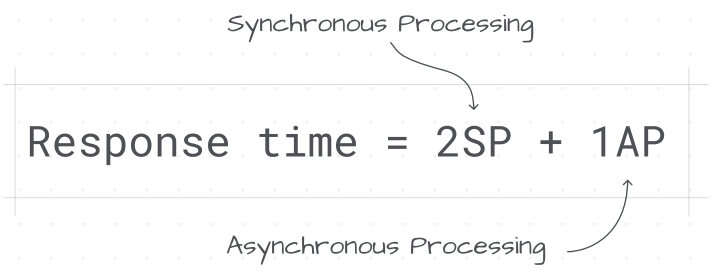
Close callbacks are associated with closing network connections or handling errors during I/O events. The event loop will then look for scheduled timers.

The loop then continues, keeping the application responsive and non-blocking.

A.3 Normal Flow Of HTTP Request In Node

When a request comes in Node.js, it is processed synchronously, with the response then undergoing a similar process. However, when a request needs to call the database, it runs asynchronously.

This means that for every request, there are 2 synchronous processes and one asynchronous process. Typically, the response time can be calculated from the formula below:



Where **SP** is Synchronous Processing and **AP** is Asynchronous Processing.

For instance, if a request takes 10ms of synchronous processing time and 10 ms of asynchronous processing time, the total response time will be:

2(10) + 10 = 30ms.

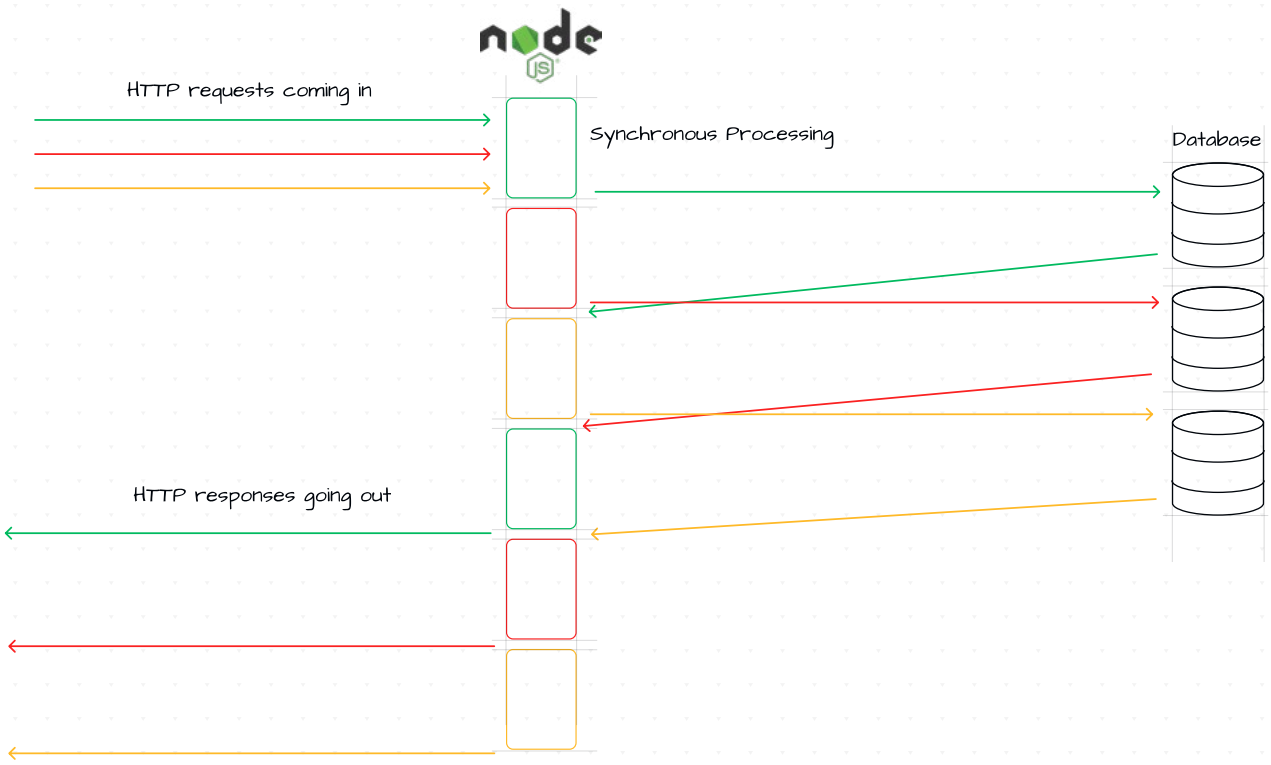
*Note: The synchronous part of the initial call and the synchronous part of the callback are independent of each other and could have different times.

Response time= call time + async time + callback time.

To calculate the total number of requests serviceable by one CPU can be calculated by:

1000ms/(10ms*2) = 50

The I/O wait is not considered because the event loop runs synchronously.



A.4 What Happens When All Requests Arrive At The Same Time?

If, for instance, a server receives three requests at once, how long it will take to process the last request?

The first request is processed while the second and third requests are queued. The second and third requests are then processed in the order they arrived, waiting for the preceding request to finish processing.

The processing time for each of the requests using the standard formula will be 30ms, 50ms and 70ms respectively, with the event loop running synchronously.

To calculate the response time for the last request, irrespective of the number of requests, you can apply the formula:

$$\text{Response time} = 2SP_x * 2 + AS_x + (SP_{x-1} * 2)$$

number of the request

Where **x** is the number of the request.

When we receive 100 requests, you can calculate how long it will take to receive any of the responses.

A possible solution to reducing this execution time is by scaling your servers based on the CPU usage: however, spawning new servers takes time and it often result in underutilized resources because that there can be available capacity in your system despite 100% utilization.

The reason for this is simple: Node.js runs on multiple threads, with a garbage collector and the CPU optimizer running separately.

This means that within Node.js, there can be a large amount of free CPU before anything starts to slow down significantly.

A.5 The Event Loop Delay

Event loop delays are measurable, meaning that developers can track when an event should fire and when it actually fired. To get an idea of how this works, you can clone this repo locally and run the code. In this repo, the loopbench.js file contains the following code:

```
const EE = require('events').EventEmitter

const defaults = {
  limit: 42,
  sampleInterval: 5
}

function loopbench(opts) {
  opts = Object.assign({}, defaults, opts)

  const timer = setInterval(checkLoopDelay, opts.sampleInterval)
  timer.unref()

  const result = new EE()

  result.delay = 0
  result.sampleInterval = opts.sampleInterval
  result.limit = opts.limit
  result.stop = clearInterval.bind(null, timer)

  let last = now()

  return result

  function checkLoopDelay() {
    const toCheck = now()
    const overLimit = result.overLimit
    result.delay = Number(toCheck - last - BigInt(result.sampleInterval))
    last = toCheck

    result.overLimit = result.delay > result.limit

    if (overLimit && !result.overLimit) {
      result.emit('unload')
    } else if (!overLimit && result.overLimit) {
      result.emit('load')
    }
  }

  function now() {
    return process.hrtime.bigint() / 1000000n
  }
}

module.exports = loopbench
```

The **example.js** file contains the code below:

```
const http = require('http')
const server = http.createServer(serve)
const loopbench = require('./')()

loopbench.on('load', function () {
  console.log('max delay reached', loopbench.delay)
})

function sleep(msec) {
  let i = 0
  const start = Date.now()
  while (Date.now() - start < msec) { i++ }
  return i
}

function serve(req, res) {
  console.log('current delay', loopbench.delay)
  console.log('overLimit', loopbench.overLimit)

  if (loopbench.overLimit) {
    res.statusCode = 503 // Service Unavailable
    res.setHeader('Retry-After', 10)
  }

  res.end()
}

server.listen(0, function () {
  const req = http.get(server.address())

  req.on('response', function (res) {
    console.log('got status code', res.statusCode)
    console.log('retry after', res.headers['retry-after'])

    setTimeout(function () {
      console.log('overLimit after load', loopbench.overLimit)
      const req = http.get(server.address())

      req.on('response', function (res) {
        console.log('got status code', res.statusCode)

        loopbench.stop()
        server.close()
      }).end()
    }, parseInt(res.headers['retry-after'], 10))
  }).end()

  setImmediate(function () {
    console.log('delay after active sleeping', loopbench.delay)
  })

  sleep(500)
})
```

The **example.js** file contains the code below:

```
max delay reached 547
delay after active sleeping 547
current delay 4
overlimit false
got staus 200
retry after undefined
overLimit after load falsecurrent delay 0
current delay 0
overLimit false
got status code 200
```

A.6 Event Loop Utilization

Event loop utilization (ELU) refers to the cumulative duration of time the event loop has been both idle and active as a high-resolution milliseconds timer. We can use it to know if there is "spare" capacity in the event loop.

ELU is a metric to monitor the amount of time spent in the event loop utilizing the CPU, and can be read straight from libuv - the C library that Node.js uses to implement the event loop.

You can compute ELU using the perf_hooks library. This will return a decimal between 0 and 1, which tells you how much of the event loop was used.

In Fastify, one of the fastest Node.js web frameworks, there is an automatically set up module called @fastify/under-pressure. You can use it to specify the max event loop delay, the memory and the event loop utilization.

So how does this package work?

When the package receives multiple requests after a certain time, the event utilization goes out of the limit at 0.98s. After this point, any request that comes in gets a response status code of 503.

Imagine having multiple requests, the event loop could have accumulated over 2 seconds of delay. A user might not find it comfortable to wait that long. In this case, you can return a response to let the user know that the server will not return to the request.

So how would this look?

To begin with, clone this repo, navigate into the thrashing directory, and find the server.js file which starts the server.

Start the server by running the following command:

```
node server.js
```

Then, in another terminal, run the command to emulate 50 connections for 10 seconds to your server:

```
npm run demo
```

You will obtain this result in the terminal:

```
> demo
> autocannon -c 50 -d 10 -t 1 --renderStatusCode http://127.0.0.1:3000

Running 10s test @ http://127.0.0.0:3000
50 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	95 ms	538 ms	1004 ms	1004 ms	539.84 ms	273.58 ms	1004 ms

Stat Min	1%	2.5%	50%	97.5%	Avg	Stdev	Max
Req/Sec	0	0	0	30	3.1	8.98	1
Bytes/Sec	0 B	0 B	0 B	5.28 KB	546 B	1.58 KB	176 B

Code	Count
200	31

From the output above, the latency is slightly above 1 second and the average request per second is 3 requests.

Let's now see what `@fastify/under-pressure` does differently. In the server-protected.js file, the maximum event loop delay is set to 200ms and the event loop utilization to 0.80.

Now start the server using the command below:

```
node server-protected.js
```

Then in another terminal, run the command to emulate the 50 connections to your server in 10 seconds. This time, you have a different result as shown below.

```
Running 10s test @ http://127.0.1:3000
50 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	5 ms	6 ms	19 ms	338 ms	14.19 ms	66.17 ms	1004 ms

Stat Min	1%	2.5%	50%	97.5%	Avg	Stdev	Max
Req/Sec	0	0	262	6443	1865.1	2157.26	16
Bytes/Sec	0 B	0 B	78.3 KB	2.02 MB	584 KB	678 KB	2.82 KB

Code	Count
200	31
503	18546

```
Req/Bytes counts sampled once per second.
# of sample: 10

109 2xx response, 18546 non 2xx response
19k requests in 10.09s, 5.84 MB read
207 errors (207 timeouts)
```

Here we can see that we got a lot more requests– 19k, compared to 527 in the first instance. Here, we got 96 successful requests compared to 31 using the unprotected server.

The function of the under-pressure package is evident by the number of 503 response statuses. The latency is also superior, with 338ms.

The server-load-aware.js file is a slight upgrade because it can even tell if the server is under pressure, offering you more control over what you want your server to do when it is under pressure and when it is not.

When we start our server load and run the demo again, this time we will obtain better numbers.

```
> demo
> autocannon -c 50 -d -t 1 --renderStatusCodes http://127.0.0.1:3000

Running 10s test @ http://127.0.1:3000
50 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	13 ms	15 ms	141 ms	603 ms	29.58 ms	91.92 ms	1012 ms

Stat Min	1%	2.5%	50%	97.5%	Avg	Stdev	Max
Req/Sec	0	0	585	2595	916.2	904.24	18
Bytes/Sec	0 B	0 B	103 KB	457 KB	161 KB	159 KB	3.17 KB

Code	Count
200	31

```
Req/Bytes counts sampled once per second.
# of sample: 10

9K request in 10.11s MB read
207 errors (207 timeouts)
```

Here, the server can handle more requests per second, compared to the previous two instances. In this case, the 200 response statuses is the highest compared to the other two instances which we looked at. The latency time is also quite low.

The biggest trade-off here is that the server sends cached data rather than returning a 503 response status. In this way, it can handle a lot more traffic and requests.

Best Practice for Event Loops

It is important to always use the event loop efficiently to ensure uninterrupted responsiveness, improved performance, maintainability and scalability.

Do not block the event loop

Move all synchronous processing outside the event loop. Consider moving them to worker threads, which are optimized to do the heavy lifting, namely, taking the strain of synchronous, computationally demanding tasks off the main thread of the Node.js event loop.

This allows your application to maintain its responsiveness and scalability while still performing computationally intensive operations.

You can check out Piscina here. It creates a pool of worker threads, which can process many tasks in parallel. It also gives you an idea of how many jobs are queuing to be processed, offering users a better view of what is happening within their server.

Deduplicate Asynchronous Calls

You can make your application faster by reducing the number of overlapping asynchronous tasks. This is where deduplication comes in, which is all about using a single unique data copy and getting rid of redundant data copies, which will still point to the used data copy.

If your application receives three requests for the same data simultaneously, they get deduplicated and only one request goes to the database for processing. We can now respond to all of them through the data that we generated earlier.
This way, your application no longer has to process data for each request as it comes in.

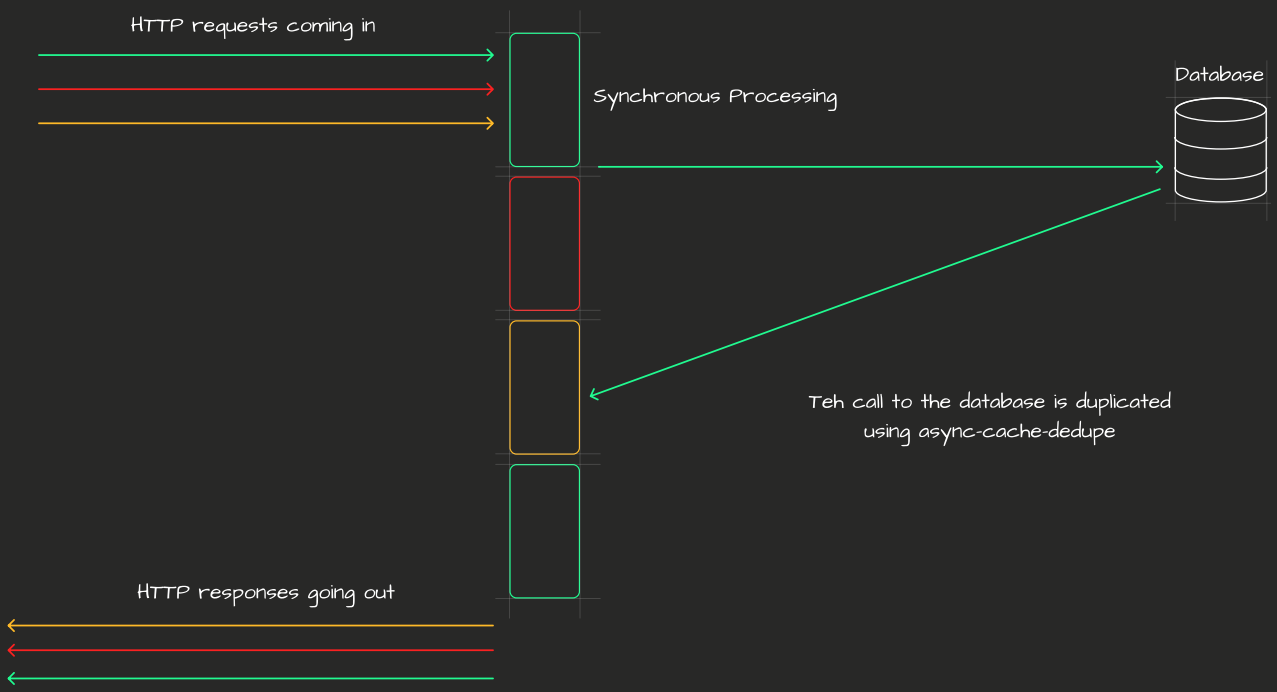
To solve this issue, you can use the async-cache-dedupe package. It is a cache for asynchronous fetching of resources with full duplication, meaning that the same resource will only be served once at any given time.

The API provides options such as `ttl`, which specifies the maximum time an entry can live. The `stale` option specifies the time after which the value is served from the cache after it has expired.

It also provides a `memory` option that defaults to storage and is compatible with Redis. The size of this memory option can also be set.

Note: Platformatic has all these packages integrated into it by default, ensuring your event loop is efficiently utilized.

Async-cache-dedupe



`async-cache-dedupe` is a cache for asynchronous fetching of resources with full deduplication. It plays a vital role in reducing response times and improving your API's efficiency with minimal effort.

`Async Cache Dedupe` stores frequently requested data in memory, which is a temporary storage location.

This means that the data your API needs can be accessed quickly without needing to repeatedly fetch this data from slower sources, like databases, every time.

Make Node

PLATFORMATIC