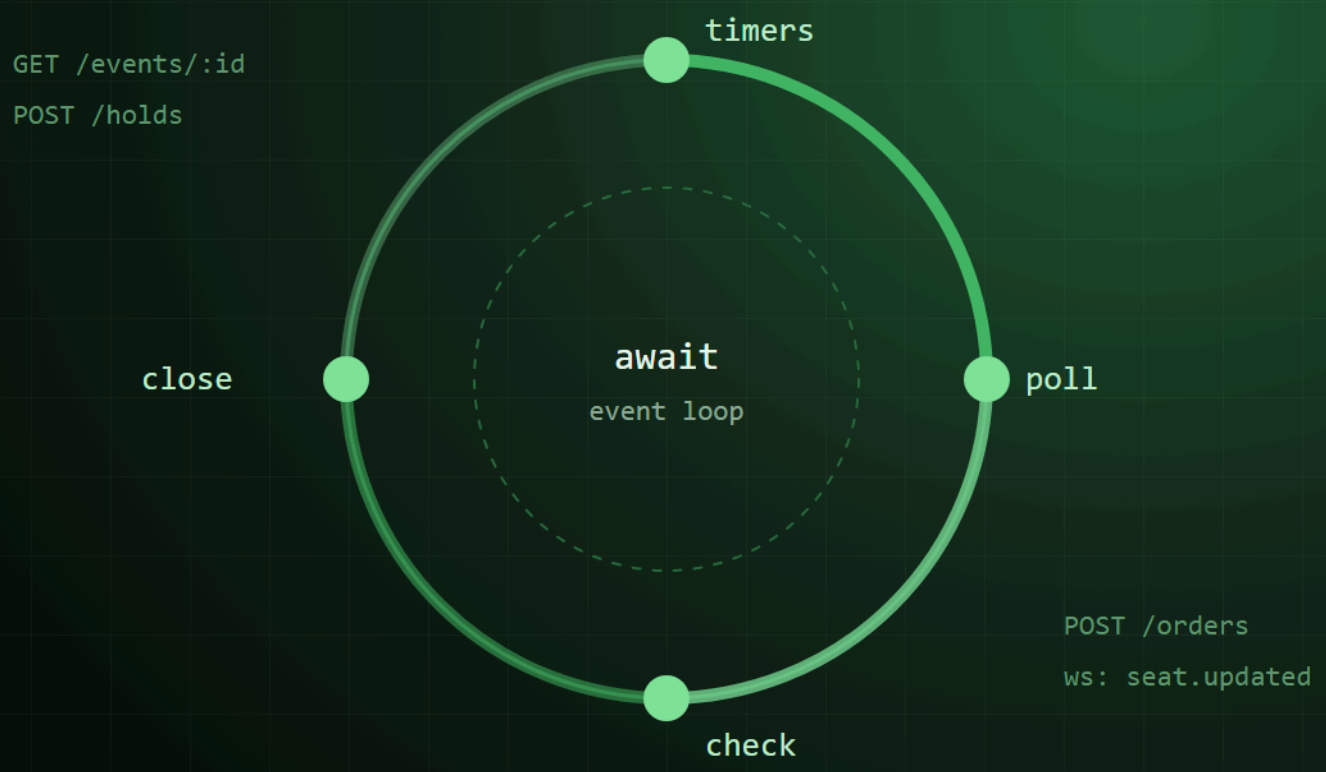


THE COMPLETE PRACTICAL GUIDE

Node.js Backend Development

APIs, Databases, and Production Deployment

Express · Fastify · PostgreSQL · Redis · Queues · Real-Time · Observability · Kubernetes

28 CHAPTERS · 4 APPENDICES · TYPESCRIPT

THENEXBOOKS

Node.js Backend Development

APIs, Databases, and Production Deployment

From the Event Loop to Flash-Sale-Safe Services in Production

First Edition

THENEXBOOKS

Node.js Backend Development
APIs, Databases, and Production Deployment

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Node.js, its ecosystem, and third-party libraries evolve quickly; always consult the official documentation for the versions you use.

Node.js is a trademark of the OpenJS Foundation. JavaScript is a trademark of Oracle and/or its affiliates. TypeScript is a trademark of Microsoft Corporation. Express, Fastify, PostgreSQL, MongoDB, Redis, Prisma, Docker, Kubernetes, OpenTelemetry, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with or endorsed by the OpenJS Foundation.

Harbor Tickets, the running example in this book, is fictional. Any resemblance to real companies or products is coincidental.

Preface

Node.js changed backend development by letting one language run on both sides of the wire, and by making non-blocking I/O the default rather than an advanced technique. A single Node.js process can hold thousands of open connections, stream data without buffering it all in memory, and talk to databases, caches, and other services concurrently — all with plain JavaScript or TypeScript.

That same model is also where most Node.js production problems come from. One accidental synchronous call blocks every request. An unhandled promise rejection crashes the process. A forgotten `await` in a transaction sells the same seat twice. Getting from "it works on my laptop" to "it survives a flash sale" requires understanding what Node.js is actually doing — and the patterns that experienced teams use to keep services fast, correct, and observable.

This book is a complete, practical guide to building backend services with Node.js — from the event loop to a production deployment with PostgreSQL, Redis, background jobs, real-time updates, observability, security, and a CI/CD pipeline. We explain *why* things work the way they do, show realistic code, and spend time on what goes wrong in production.

Who This Book Is For

- **Frontend developers** who know JavaScript and want to build and own the backend.
- **Backend developers** coming from Java, C#, Python, PHP, or Go who need to become productive in Node.js quickly.
- **Node.js developers** who can build an Express app but want the full production picture: data integrity, queues, testing, observability, and deployment.
- **Tech leads** designing Node.js services and setting standards for a team.

What You Will Learn

- How the event loop, libuv, and asynchronous I/O really work — and how to avoid blocking them.
- Modern TypeScript, ES modules, project setup, promises, streams, and event emitters for backend work.
- HTTP APIs with `node:http`, Express, and Fastify; validation with Zod and JSON Schema; OpenAPI documentation.
- Authentication and authorization, WebSockets and Server-Sent Events, GraphQL, and gRPC.
- PostgreSQL with `pg`, Prisma, Drizzle, and Knex; transactions and row locks; MongoDB; Redis caching, rate limiting, and locks.
- Application structure, background jobs with BullMQ, event-driven design, the outbox pattern, webhooks, and idempotency.
- Testing, logging with Pino, metrics and tracing with OpenTelemetry, profiling, worker threads, security, containers, Kubernetes, and CI/CD.

- Two end-to-end case studies: a flash-sale-safe checkout API and an AI assistant endpoint with streaming responses.

How This Book Is Organized

Part I — Node.js Foundations (Chapters 1–5) explains the runtime and the language features backend code relies on. **Part II — Building HTTP APIs** (Chapters 6–11) builds APIs from raw `node:http` to Express, Fastify, authentication, and real-time endpoints. **Part III — Data and Persistence** (Chapters 12–16) connects services to PostgreSQL, MongoDB, and Redis without losing data integrity. **Part IV — Architecture and Background Work** (Chapters 17–21) covers structure, queues, events, GraphQL, gRPC, and testing. **Part V — Production Readiness** (Chapters 22–26) addresses logging, observability, performance, security, and deployment. **Part VI — Case Studies** (Chapters 27–28) brings everything together. The appendices provide a cheat sheet, a service template, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Harbor Tickets

Throughout the book we build **Harbor Tickets**, a fictional event-ticketing marketplace. Organizers create venues and events with seating maps; customers hold seats, check out, pay, and receive tickets with QR codes; staff check tickets in at the door. Ticketing is a demanding domain for a backend: flash sales bring thousands of buyers to the same seats at the same moment, holds must expire on time, payments and webhooks must be idempotent, and seat maps must update in real time. Those are exactly the problems Node.js services face in production.

Conventions Used in This Book

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake or a behavior that can cause bugs, data loss, or outages in production.

Code appears in `monospace`. Examples use TypeScript with ES modules unless stated otherwise and target current LTS versions of Node.js. Features that are experimental in some Node.js versions are labeled as such.

Contents

Preface

PART I — NODE.JS FOUNDATIONS

1 Why Node.js for Backend Development

2 The Event Loop, Asynchronous I/O, and libuv

3 Modern JavaScript and TypeScript for Backend Developers

4 Modules, Packages, and Project Setup

5 Async Patterns: Promises, async/await, EventEmitter, and Streams

PART II — BUILDING HTTP APIS

6 HTTP Fundamentals with node:http and the Request/Response Lifecycle

7 Express in Depth: Routing, Middleware, and Error Handling

8 Fastify: Schemas, Plugins, Hooks, and Serialization

9 REST API Design, Validation, and OpenAPI

10 Authentication and Authorization

11 Real-Time APIs: WebSockets, Server-Sent Events, and Socket.IO

PART III — DATA AND PERSISTENCE

12 PostgreSQL with node-postgres: Pools, Queries, and Parameters

13 ORMs and Query Builders: Prisma, Drizzle, and Knex

14 Transactions, Concurrency Control, Migrations, and Data Integrity

15 MongoDB with the Official Driver and Mongoose

16 Caching, Rate Limiting, and Distributed Locks with Redis

PART IV — ARCHITECTURE AND BACKGROUND WORK

17 Structuring Node.js Applications: Layers, Modules, and Dependency Injection

18 Background Jobs, Queues, and Scheduling

19 Event-Driven Services: Messaging, the Outbox Pattern, Webhooks, and Idempotency

20 GraphQL and gRPC in Node.js

21 Testing Node.js Services: node:test, Vitest, Supertest, and Testcontainers

PART V — PRODUCTION READINESS

22 Configuration, Structured Logging with Pino, and Error Handling in Production

23 Observability: Metrics, Tracing, and OpenTelemetry

24 Performance: Profiling, Worker Threads, Clustering, and Memory Leaks

25 Security for Node.js APIs

26 Containers, Kubernetes, CI/CD, and Graceful Shutdown

PART VI — CASE STUDIES

27 Case Study: Building the Harbor Tickets Checkout API End to End

28 Case Study: Adding an AI Assistant Endpoint with Streaming Responses

APPENDICES

A Node.js and npm Cheat Sheet

B Service Template and Reference

C Production Checklist

D Glossary

PART I

Node.js Foundations

The runtime under your code: the event loop, asynchronous I/O, modern TypeScript, modules, and the async building blocks — promises, event emitters, and streams — that every Node.js service is made of.

1 · Why Node.js for Backend Development
2 · The Event Loop and libuv
3 · Modern JavaScript and TypeScript
4 · Modules, Packages, and Project Setup
5 · Async Patterns

CHAPTER 1 · PART I — NODE.JS FOUNDATIONS

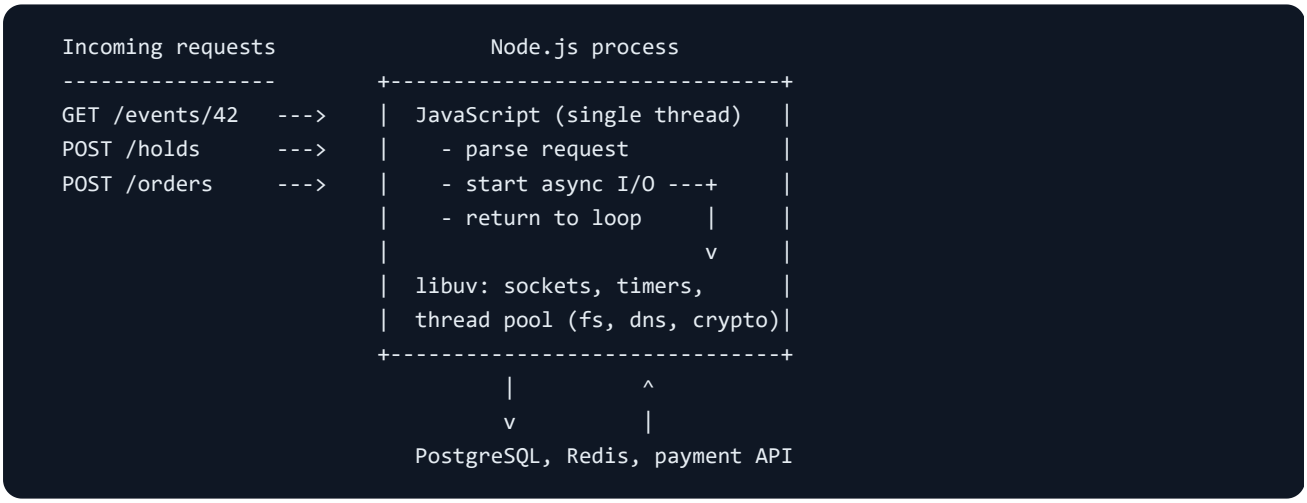
Why Node.js for Backend Development

Before we write a single route, we need an honest answer to a simple question: why build a backend in Node.js at all? This chapter explains what Node.js actually is, where it shines, where it struggles, and introduces Harbor Tickets, the event-ticketing marketplace we will build and harden throughout the rest of the book.

What Node.js Actually Is

Node.js is a runtime that executes JavaScript outside the browser. That one-sentence definition is accurate, but it hides the part that matters most for backend work. Node.js combines three things: the V8 JavaScript engine (the same engine that powers Chromium-based browsers), a C library called libuv that provides an event loop and asynchronous access to the operating system, and a standard library of modules such as `node:http`, `node:fs`, `node:crypto`, and `node:stream` that expose networking, files, cryptography, and data streaming to JavaScript code.

The design decision that defines Node.js is that your JavaScript runs on a single main thread, and almost every input/output operation is non-blocking. When your code asks the database for a row or asks the network for an HTTP response, Node.js does not park a thread and wait. It registers interest in the result, returns control to the event loop, and continues handling other work. When the result arrives, the event loop schedules your callback, promise continuation, or `await` resumption. We will dissect this machinery in detail in Chapter 2, but you need the high-level picture now, because it explains almost every strength and weakness we discuss below.



Contrast this with the classic thread-per-request model used by many Java, C#, and PHP deployments. There, each in-flight request typically occupies an operating system thread (or a process). While a request waits for the database, its thread sits idle but still consumes memory for its stack and adds scheduling overhead. Modern runtimes have introduced lighter-weight alternatives,

such as virtual threads on the JVM or async/await in .NET and Python, so the gap is narrower than it used to be. Still, Node.js was built around non-blocking I/O from the start, and its entire ecosystem assumes it. You will rarely find a popular Node.js database driver or HTTP client that blocks the thread while waiting on the network.

A Minimal Server, Without Frameworks

To make this concrete, here is a complete HTTP server using only the built-in `node:http` module. It serves a single health-check endpoint for Harbor Tickets. We use TypeScript throughout this book; Chapter 4 shows how to set up the compiler, and Chapter 3 covers the language features you will see here.

src/server.ts

```
import { createServer } from "node:http";

const port = Number(process.env.PORT ?? 3000);

const server = createServer((req, res) => {
  if (req.method === "GET" && req.url === "/health") {
    res.writeHead(200, { "content-type": "application/json" });
    res.end(JSON.stringify({ status: "ok", service: "harbor-api" }));
    return;
  }

  res.writeHead(404, { "content-type": "application/json" });
  res.end(JSON.stringify({ error: "not_found" }));
});

server.listen(port, () => {
  console.log(`harbor-api listening on http://localhost:${port}`);
});
```

There is no container to configure, no application server to install, and no framework required. The process *is* the server. That directness is part of the appeal: you can reason about everything between the socket and your handler. In practice we will use frameworks such as Express (Chapter 7) and Fastify (Chapter 8) because they solve routing, validation, and error handling well, but it is valuable to know that underneath them sits this same small API. Chapter 6 explores it thoroughly.

Where Node.js Shines

Choosing a backend runtime is a trade-off, not a popularity contest. Node.js is a strong fit for a specific and very common class of services. Let us look at why.

I/O-Bound Workloads

Most backend services spend the majority of their time waiting: waiting on a database, a cache, a message broker, a third-party API, or a slow client uploading a request body. The CPU work per request, such as parsing JSON, checking a token, and shaping a response, is often small compared to

the waiting. This is called an I/O-bound workload, and it is exactly what the non-blocking model was designed for. A single Node.js process can hold many concurrent connections open because an idle connection costs little more than some memory for its socket and buffers.

Consider the Harbor Tickets endpoint that returns an event page. It fetches the event from PostgreSQL, reads remaining seat counts from Redis, and asks the promotions service whether a discount applies. None of those calls depend on each other, so we can issue them concurrently:

src/events/events.service.ts

```
import type { Pool } from "pg";
import type { Redis } from "ioredis";

export interface EventSummary {
  id: string;
  title: string;
  startsAt: string;
  seatsRemaining: number;
  promoAvailable: boolean;
}

export async function getEventSummary(
  db: Pool,
  cache: Redis,
  promoBaseUrl: string,
  eventId: string,
): Promise<EventSummary | null> {
  const [eventResult, remaining, promoResponse] = await Promise.all([
    db.query(
      "SELECT id, title, starts_at FROM events WHERE id = $1",
      [eventId],
    ),
    cache.get(`event:${eventId}:seats_remaining`),
    fetch(`${promoBaseUrl}/events/${encodeURIComponent(eventId)}/promo`, {
      signal: AbortSignal.timeout(500),
    }).catch(() => null),
  ]);

  const row = eventResult.rows[0];
  if (!row) return null;

  return {
    id: row.id,
    title: row.title,
    startsAt: row.starts_at.toISOString(),
    seatsRemaining: Number(remaining ?? 0),
    promoAvailable: promoResponse?.ok ?? false,
  };
}
```

Three network calls are in flight at the same time, and while they are pending the same process can accept and begin handling other requests. There is no thread pool to size for this and no explicit synchronization. Notice also the small defensive details we will keep repeating in this book: a

parameterized SQL query (`$1`) instead of string concatenation, a timeout on the outbound HTTP call using `AbortSignal.timeout`, and a fallback when the non-essential promo service fails. Global `fetch` is available in Node.js 18 and later, so we do not need an extra HTTP client for simple calls.

One Language Across the Stack

If your team already writes TypeScript for the browser, Node.js lets you share code, types, and tooling with the backend. A validation schema for creating a seat hold can be defined once and used by both the checkout page and the API. Engineers can move between frontend and backend work without switching language, package manager, linter, or test runner. This is not a technical superpower, but it has real organizational value: fewer contexts to maintain, shared code review norms, and a larger pool of people who can fix a bug at two in the morning.

packages/contracts/src/holds.ts

```
import { z } from "zod";

export const CreateHoldRequest = z.object({
  eventId: z.string().uuid(),
  seatIds: z.array(z.string().uuid()).min(1).max(10),
});

export type CreateHoldRequest = z.infer<typeof CreateHoldRequest>;

export interface HoldResponse {
  holdId: string;
  seatIds: string[];
  expiresAt: string; // ISO 8601 timestamp
}
```

The frontend imports `CreateHoldRequest` to validate the form before submission; the backend imports the same schema to validate the request body. When a product manager decides that customers may hold up to eight seats instead of ten, one change updates both sides, and the TypeScript compiler flags every place that needs attention. Chapter 9 covers validation with Zod in depth, and Chapter 4 shows how to structure a workspace so that packages like `contracts` can be shared.

A Large, Practical Ecosystem

The npm registry hosts an enormous number of packages. For backend work, the important point is not the raw count but the fact that mature, well-maintained libraries exist for nearly every production concern: HTTP frameworks (Express, Fastify), database drivers (`pg`, the official MongoDB driver), ORMs and query builders (Prisma, Drizzle ORM, Knex), Redis clients (ioredis, redis), job queues (BullMQ), logging (Pino), telemetry (OpenTelemetry JS), and testing (Vitest, Jest, Supertest, Testcontainers). We will use all of them in this book.

The flip side is that the ecosystem's breadth requires judgment. Anyone can publish a package, and small packages can pull in long chains of transitive dependencies. Every dependency is code you are choosing to run in production with access to your secrets. We discuss how to evaluate and audit

dependencies in Chapter 4 and supply-chain security in Chapter 25.

> **WARNING**
>
> Do not judge a package only by its download count or star count. Check when it was last released, whether issues get answered, how many dependencies it brings along, and whether it has a clear security policy. A small, unmaintained package in your checkout path is a liability, no matter how popular it once was.

Real-Time and Streaming

Because Node.js is comfortable holding many long-lived connections, it is a natural fit for WebSockets, Server-Sent Events (SSE), and streaming responses. Harbor Tickets needs a live seat map: when someone in Berlin holds seats 14 and 15 in row C, a customer looking at the same event in Lisbon should see those seats turn grey within a moment. We build that in Chapter 11. Streams are also first-class in Node.js, so you can pipe a large CSV report to a client row by row without loading it all into memory, a technique we use for organizer reports in Chapter 5.

Fast Feedback During Development

Node.js processes start quickly, and recent versions include a built-in `--watch` mode that restarts your server when files change. Combined with TypeScript tooling that transpiles on the fly, the edit-and-retry loop is short. This matters more than it sounds: a short loop encourages small experiments, and small experiments catch mistakes early.

Where Node.js Struggles

A book that only lists strengths would not be much help when you are designing a real system. The same single-threaded design that makes I/O cheap makes some other things hard.

CPU-Bound Work Blocks Everything

If your JavaScript runs a long synchronous computation, such as resizing an image, generating a large PDF, hashing a password with a very high cost factor synchronously, or parsing a multi-megabyte JSON document, the event loop cannot do anything else during that time. Every other request on that process waits. This is the single most important performance fact about Node.js, and the following code shows the mistake in its simplest form:

src/tickets/bad-report.ts (do not do this)

```
import { createServer } from "node:http";

function slowChecksum(input: string, rounds: number): number {
  let hash = 0;
  for (let r = 0; r < rounds; r++) {
    for (let i = 0; i < input.length; i++) {
      hash = (hash * 31 + input.charCodeAt(i)) | 0;
```

```
    }
  }
  return hash;
}

createServer((req, res) => {
  if (req.url === "/report") {
    // Synchronous CPU work: blocks every other request on this process.
    const value = slowChecksum("harbor-tickets", 50_000_000);
    res.end(String(value));
    return;
  }
  res.end("ok");
}).listen(3000);
```

While one client waits for `/report`, a second client requesting any other URL, even a trivial health check, also waits. Under load balancer health checks, this can cause a perfectly healthy process to be marked as dead and restarted. The remedies are well established: move CPU-heavy work to `worker_threads` (Chapter 24), to a background job processed by a separate worker process (Chapter 18), or to a service written in a language better suited for that computation. What you must not do is ignore it.

> **NOTE**
>
> Many operations that look CPU-heavy are already handled well by Node.js. For example, the asynchronous versions of `crypto.scrypt` and `crypto.pbkdf2`, file system calls, and zlib compression run on libuv's thread pool rather than the main thread. The danger is *your* synchronous JavaScript loops and the synchronous `*Sync` variants of core APIs used inside request handlers.

Single-Process Memory and Cores

One Node.js process runs JavaScript on one core. A server with many cores needs multiple processes to use them, either through the `cluster` module, through a process manager, or, most commonly today, by running several container replicas behind a load balancer. This is not hard, but it changes how you think about state. Anything stored in a process's memory, such as an in-memory cache or a map of connected WebSocket clients, is invisible to the other processes. Harbor Tickets therefore keeps shared state in PostgreSQL and Redis, never in module-level variables. Chapters 16 and 26 return to this.

The Cost of Flexibility

JavaScript is permissive. Without discipline, a Node.js codebase can accumulate inconsistent error handling, implicit type conversions, and promises that nobody awaits. TypeScript in strict mode, a linter, and a few team conventions solve most of this, which is why this book uses TypeScript by

default. Node.js also imposes no application architecture. Unlike opinionated frameworks in other ecosystems, Express gives you a router and middleware and leaves the rest to you. That freedom is excellent for small services and risky for large ones; Chapter 17 offers a concrete structure.

Comparing Node.js With Other Backend Stacks

Many readers come from another backend ecosystem. The following table is a qualitative orientation, not a benchmark. Every row has exceptions, and every modern stack can build a successful ticketing system.

Concern	Node.js	Typical alternative
Concurrency model	Event loop, async I/O	Threads, virtual threads, goroutines
CPU-heavy work	Needs workers or offloading	Often fine in-process (Go, JVM)
Language sharing with UI	Same language as browser	Separate language
Startup time	Fast	Varies by runtime
Type safety	Via TypeScript (compile-time)	Built into language (Java, C#, Go)
Architecture guidance	Minimal; team decides	Often strong (Spring, ASP.NET)

If you are coming from Java or C#, the biggest adjustment is that there is no request thread you can block. You cannot call a synchronous database driver and expect other requests to proceed. Everything that touches the network returns a promise. If you are coming from Go, you will find promises and `async/await` less explicit than goroutines and channels, but the underlying idea of cheap concurrency for I/O is similar. If you are coming from PHP, the shift is that your process lives across requests: module-level state persists, connection pools are reused, and a memory leak accumulates over hours instead of disappearing at the end of each request. If you are coming from Python, the async model will feel familiar if you have used asyncio, with the difference that in Node.js the entire ecosystem is async by default.

> **TIP**
>
> When your team debates runtimes, write down the three or four hardest requirements of the actual system, such as "no double-selling during flash sales" or "stream PDF tickets to email," and evaluate each option against those. Abstract comparisons produce heat; concrete requirements produce decisions.

Meet Harbor Tickets

Throughout this book, we build **Harbor Tickets**, a fictional event-ticketing marketplace. We chose ticketing deliberately because it contains nearly every hard problem a backend developer meets in production, wrapped in a domain that is easy to understand. Everyone has bought a concert or

conference ticket, and everyone has felt the frustration of a sold-out event.

The Domain

Organizers are the tenants of the platform. Each organizer creates **Venues** with seating **Sections** and **Seats**, then publishes **Events** at those venues. Customers, who are **Users**, browse events and select seats. Selecting seats creates a short-lived **Seat Hold**, perhaps ten minutes long, that reserves those seats while the customer checks out. Checkout creates an **Order** with one or more **OrderItems**, applies an optional **PromoCode**, and sends the customer to a fictional payment provider. When payment succeeds, the system issues **Tickets** with QR codes and delivers them by email. At the venue door, staff scan the QR codes to check attendees in. Behind the scenes, the platform handles **Refunds**, pays organizers through **Payouts**, and sends **Notifications**.

```
Organizer 1---* Venue 1---* Section 1---* Seat
    |                                      |
    1                                      |
    *                                      |
  Event 1---* SeatHold *---* Seat ----------+
    |
    1
    *
  Order 1---* OrderItem ---> Seat
    |   \
    |    *---1 PromoCode
    1
    *
  Payment ---> Refund        Ticket (per OrderItem, QR code)
```

The Hard Problems

A ticketing platform looks simple until a popular artist announces a tour. Here are the problems we will solve, and the chapters where they appear:

- **Flash sales without double-selling.** Thousands of buyers may try to hold the same front-row seats in the same second. Exactly one of them may win each seat. We use database constraints, row locks, and Redis-based coordination (Chapters 14, 16, and 27).
- **Idempotent payments and webhooks.** Networks fail, clients retry, and payment providers deliver the same webhook more than once. Charging a customer twice or issuing duplicate tickets is unacceptable. We design idempotency keys and deduplication (Chapters 19 and 27).
- **Expiring holds.** A hold that is never paid for must release its seats on time. That requires background jobs and scheduling (Chapter 18).
- **Real-time seat maps.** Customers should see seats disappear as others hold them. We push updates over WebSockets and Server-Sent Events (Chapter 11).
- **Ticket delivery.** Generating PDF tickets and sending email must not slow down checkout. We move that work to queues (Chapter 18).

- **Organizer reports.** Organizers want sales exports that may contain many rows. We stream them instead of building huge arrays in memory (Chapter 5).

Money Is an Integer

One rule applies everywhere in Harbor Tickets: money is stored as an integer number of minor units, together with a currency code. Twenty-five euros is `amountCents: 2500` with `currency: "EUR"`. JavaScript numbers are IEEE 754 double-precision floats, so decimal fractions such as 0.1 and 0.2 cannot be represented exactly, and adding them produces rounding artifacts. Integers are exact up to `Number.MAX_SAFE_INTEGER`, which is far larger than any realistic ticket price in minor units.

src/shared/money.ts

```
export type CurrencyCode = "EUR" | "USD" | "GBP";

export interface Money {
  readonly amountCents: number;
  readonly currency: CurrencyCode;
}

export function money(amountCents: number, currency: CurrencyCode): Money {
  if (!Number.isSafeInteger(amountCents)) {
    throw new RangeError(`amountCents must be a safe integer: ${amountCents}`);
  }
  return { amountCents, currency };
}

export function addMoney(a: Money, b: Money): Money {
  if (a.currency !== b.currency) {
    throw new Error(`Currency mismatch: ${a.currency} vs ${b.currency}`);
  }
  return money(a.amountCents + b.amountCents, a.currency);
}

export function formatMoney(m: Money, locale = "en-US"): string {
  return new Intl.NumberFormat(locale, {
    style: "currency",
    currency: m.currency,
  }).format(m.amountCents / 100);
}
```

We only divide by 100 at the very edge, when formatting for display, and even that assumes a currency with two minor-unit digits. A production system supporting currencies with zero or three minor-unit digits would look up the exponent per currency. The important habit is that arithmetic, storage, and comparisons all happen on integers.

The Codebase Layout

The Harbor Tickets backend lives in a TypeScript project called `harbor-api`, organized by business module rather than by technical layer. Each module owns its routes, services, and data access for one part of the domain:

```
harbor-api/
  package.json
  tsconfig.json
  src/
    app.ts              # builds the HTTP app, registers modules
    server.ts           # starts listening, handles shutdown
    config.ts           # typed configuration from environment
    shared/             # money, errors, logging, db pool
    events/             # venues, sections, seats, events
    holds/              # seat holds and expiry
    orders/             # orders, order items, promo codes
    payments/           # payment provider, webhooks, refunds
    tickets/            # ticket issuing, QR codes, check-in
    notifications/      # email and push delivery
  test/
  migrations/
```

We start small and let the structure grow. In early chapters, a module may be a single file. By Chapter 17 you will see a fuller layered design with dependency injection, and by Chapter 27 we assemble the complete flash-sale-safe checkout.

A First Taste: The Seat Hold Endpoint

To connect the ideas in this chapter, let us sketch the endpoint that creates a seat hold. It is deliberately simplified; later chapters make each part robust. Even this sketch shows why the non-blocking model suits the domain: the handler spends almost all its time waiting on PostgreSQL, and while it waits, the process serves other customers.

src/holds/holds.service.ts

```
import type { Pool } from "pg";
import { randomUUID } from "node:crypto";

const HOLD_MINUTES = 10;

export class SeatUnavailableError extends Error {
  constructor(public readonly seatIds: string[]) {
    super(`Seats unavailable: ${seatIds.join(", ")}`);
    this.name = "SeatUnavailableError";
  }
}

export async function createHold(
  db: Pool,
  userId: string,
  eventId: string,
  seatIds: string[],
): Promise<{ holdId: string; expiresAt: Date }> {
  const holdId = randomUUID();
  const expiresAt = new Date(Date.now() + HOLD_MINUTES * 60_000);
  const client = await db.connect();
```

```
  try {
    await client.query("BEGIN");
    await client.query(
      `INSERT INTO seat_holds (id, user_id, event_id, expires_at)
       VALUES ($1, $2, $3, $4)`,
      [holdId, userId, eventId, expiresAt],
    );
    // A unique index on (event_id, seat_id) for active holds and sold
    // seats makes a concurrent double-hold fail at the database.
    await client.query(
      `INSERT INTO seat_hold_items (hold_id, event_id, seat_id)
       SELECT $1, $2, unnest($3::uuid[])`,
      [holdId, eventId, seatIds],
    );
    await client.query("COMMIT");
    return { holdId, expiresAt };
  } catch (err) {
    await client.query("ROLLBACK");
    if (isUniqueViolation(err)) throw new SeatUnavailableError(seatIds);
    throw err;
  } finally {
    client.release();
  }
}

function isUniqueViolation(err: unknown): boolean {
  return (
    typeof err === "object" &&
    err !== null &&
    "code" in err &&
    (err as { code?: string }).code === "23505"
  );
}
```

Several principles are already visible. The database, not the application, is the final arbiter of whether a seat is available, because a unique index cannot be fooled by two processes racing each other. The transaction is always either committed or rolled back, and the pooled connection is always released in `finally`. Errors are translated into domain errors that an HTTP layer can map to a `409 Conflict`, while unexpected errors propagate unchanged. PostgreSQL's error code `23505` identifies a unique-constraint violation. Chapter 12 covers pooling and parameters, and Chapter 14 revisits this function with row locks and isolation levels.

Here is how an Express route might expose this service. Express 5 forwards a rejected promise from an async handler to the error middleware automatically, so we can simply `await` and let unexpected errors flow onward. In Express 4 you needed a small wrapper to achieve the same effect.

src/holds/holds.routes.ts

```
import { Router } from "express";
import type { Pool } from "pg";
import { CreateHoldRequest } from "@harbor/contracts";
import { createHold, SeatUnavailableError } from "./holds.service.js";
```

```
export function holdsRouter(db: Pool): Router {
  const router = Router();

  router.post("/holds", async (req, res) => {
    const parsed = CreateHoldRequest.safeParse(req.body);
    if (!parsed.success) {
      res.status(400).json({ error: "invalid_request" });
      return;
    }

    const userId = res.locals.userId as string; // set by auth middleware
    try {
      const hold = await createHold(
        db,
        userId,
        parsed.data.eventId,
        parsed.data.seatIds,
      );
      res.status(201).json({
        holdId: hold.holdId,
        seatIds: parsed.data.seatIds,
        expiresAt: hold.expiresAt.toISOString(),
      });
    } catch (err) {
      if (err instanceof SeatUnavailableError) {
        res.status(409).json({ error: "seats_unavailable" });
        return;
      }
      throw err; // Express 5 passes this to error-handling middleware
    }
  });

  return router;
}
```

What Goes Wrong in Production

Node.js services rarely fail in exotic ways. They fail in a handful of predictable ones, and knowing them from the start will shape how you write code in every later chapter.

1. **Blocked event loop.** A synchronous loop, a huge `JSON.parse`, or a synchronous file read inside a handler stalls every request on the process. Symptoms are latency spikes across unrelated endpoints and failing health checks. Chapter 2 shows how to measure event-loop delay, and Chapter 24 shows how to profile.
2. **Unhandled promise rejections.** A promise that rejects without a handler terminates the process by default in recent versions of Node.js. That is the right default, since silently ignoring errors is worse, but it means a missing `await` or `catch` can crash production.
3. **Unbounded concurrency.** Because starting async work is cheap, it is easy to start too much at once, for example firing ten thousand database queries with `Promise.all` over a large array. The pool queues them, memory grows, and latency explodes. Chapter 5 shows how to limit