Modern Full Stack Frameworks

A Practical Handbook for Students, Hackathons, and Startups

Shashank Jagannatham

2025

Contents

Chapter 1

Introduction

Software development has changed dramatically over the past decade. In the early 2010s, developers often combined jQuery for frontend interactivity, PHP or Java servlets for backend logic, and MySQL for data. Each piece worked in isolation, often requiring manual setup, wiring, and endless debugging of environments.

By 2025, the picture looks very different. Today, developers have access to **modern full stack frameworks** that reduce boilerplate, improve scalability, and allow small teams to build production-grade systems quickly. These frameworks are not just convenient—they are strategic tools for hackathons, startups, and enterprise developers alike.

1.1 Why Full Stack Matters

A "full stack" application connects three critical layers:

1. **Frontend** – What users see and interact with (React, Next.js).
2. **Backend** – Logic, authentication, APIs (Node.js, Express).
3. **Database** – Persistent storage (PostgreSQL, Prisma).

Supporting technologies like Redis, Docker, and cloud platforms ensure the system is fast, consistent, and deployable anywhere.

1.2 The Evolution of Stacks

Yesterday: MERN (MongoDB, Express, React, Node) was popular in 2015. It worked but required a lot of manual configuration, and MongoDB's flexibility sometimes caused data consistency issues.

Today: A modern stack often looks like Next.js (frontend + light backend) + Express (API layer) + Prisma (DB access) + Postgres (data) + Redis (cache) + Docker (containerization). These pieces form a cohesive workflow that lets you move from idea to deploy rapidly.

1.3 A Story: Building at a Hackathon

Imagine you are at a 48-hour hackathon, trying to build a fintech app to send money across borders. Time is short, and every hour counts.

- With React alone, you'd spend hours configuring routing and SEO. Next.js solves this in minutes.
- With raw SQL, you'd be writing queries from scratch. Prisma generates type-safe queries automatically.
- Without Redis, your app would repeatedly call external APIs. With caching, you save costs and speed up responses.
- Without Docker, "it works on my machine" becomes a nightmare when deploying. Containers guarantee consistency.

By the end of the hackathon, teams using modern stacks spend more time innovating—and less time fighting boilerplate.

1.4 Who This Book is For

This book is designed for:

- **Students** learning how real-world applications are built.
- **Hackathon builders** who need to go from idea to demo in a weekend.
- **Startup developers** building MVPs that can scale.

Each chapter offers:

- Clear explanations of the technology.
- Visual diagrams.
- Hands-on exercises.
- Mini projects to practice.

1.5 Looking Ahead

In the following chapters, we'll break down each layer of the modern stack:

- **Frontend:** React to Next.js.
- **Backend:** Node.js with Express.
- **Database:** PostgreSQL with Prisma.
- **Caching:** Redis.
- **Deployment:** Docker and managed cloud platforms.

By the end, you'll not only understand how these tools work individually, but also how to stitch them together into one cohesive, production-ready system.

Key Takeaway: Modern full stack frameworks are about speed, reliability, and focus. They help you spend less time on setup and more time on building what matters.

Chapter 2

Frontend: React to Next.js

2.1 Concepts

React is a UI library that focuses only on the “view” layer of applications. It provides reusable components, props for passing data, and state for dynamic changes. However, React by itself is limited:

- No built-in routing (you need libraries like React Router).
- SEO challenges due to client-side rendering.
- Configuration heavy (build tools like Vite or CRA are needed).

Next.js builds on React and addresses these gaps:

- File-based routing: pages are created by placing files in the `pages/` directory.
- Server-Side Rendering (SSR), Static Site Generation (SSG), and Incremental Static Regeneration (ISR).
- API routes: build backend endpoints inside the same project.
- Server Components (default) vs Client Components (with “use client”).
- Optimizations like Image component, Middleware, and production defaults.

2.2 Visual Diagram

$$\begin{tikzcd}
\boxed{\text{React (UI Library)}} \arrow[r] & \boxed{\text{Next.js (Full Framework)}}
\end{tikzcd}$$

2.3 Key Differences

- React renders only in the browser; Next.js can render on both server and client.
- React requires external tools for routing; Next.js has built-in routing.
- React projects often need separate backends; Next.js can serve APIs from the same codebase.

2.4 Hands-On Exercise

1. Create a new app: `npx create-next-app myapp`
2. Add a login page: create `pages/login.js` with a simple form.
3. Add a dashboard page: create `pages/dashboard.js` that fetches mock data server-side.
4. Run the app: `npm run dev` and open `http://localhost:3000`.

2.5 Mini Project

Build a two-page app:

- A login page (client component) with a simple username form.
- A dashboard page (server component) that fetches current cryptocurrency prices from a public API.

This project shows how Next.js combines frontend and backend features into one cohesive workflow.

Chapter 3

Backend: Node.js and Express

3.1 Concepts

Node.js allows us to run JavaScript outside the browser, on the server. By default, it doesn't provide routing or middleware. That's where Express comes in.

Express is a minimal and flexible Node.js web application framework. It provides:

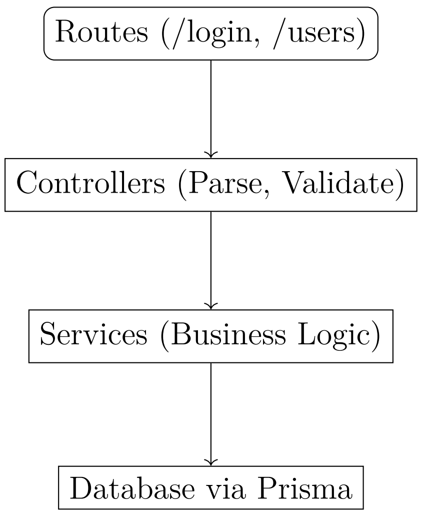
- Routing (defining endpoints such as GET /login, POST /users).
- Middleware (authentication, validation, logging, rate limiting).
- Integration with databases and ORMs like Prisma.

3.2 API Layering

A clean architecture for APIs often looks like this:

- **Routes** – define URLs and map them to controllers.
- **Controllers** – parse requests, validate data, and call services.
- **Services** – contain the business logic.
- **Database Layer** – Prisma functions that handle persistence.

3.3 Visual Diagram



3.4 Hands-On Exercise

1. Initialize an Express app: `npm install express`.
2. Add a health check route:

```
import express from 'express';
const app = express();

app.get('/health', (_req, res) => res.json({ ok: true }));

app.listen(8080, () => console.log('API running on 8080'));
```

3. Add middleware for JSON parsing and CORS.
4. Implement JWT authentication with access and refresh tokens.

3.5 JWT Authentication Flow

JWT (JSON Web Token) is used to securely transmit user identity between frontend and backend.

1. User logs in with email/password.
2. Backend verifies credentials, generates an **access token** (short-lived) and a **refresh token** (long-lived).
3. Access token is sent in the Authorization header (Bearer scheme).
4. Refresh token is stored in a secure httpOnly cookie.
5. Middleware checks access token; if expired, frontend calls refresh endpoint to get a new one.

3.6 Mini Project

Build a small Express API with two endpoints:

- POST /auth/login – returns JWT tokens after verifying a dummy user.
- GET /profile – returns user profile only if a valid access token is provided.

This project will demonstrate authentication and clean API layering.

Chapter 4

Database: PostgreSQL and Prisma

4.1 Concepts

PostgreSQL is a powerful open-source relational database. It uses tables, rows, and SQL queries to manage structured data. While SQL is expressive, writing raw queries in application code can be verbose and error-prone.

Prisma solves this by providing:

- A schema definition file (`schema.prisma`) to model tables.
- A type-safe client with autocomplete for Node.js/TypeScript.
- Migrations to version and update the database schema.
- Prisma Studio, a UI to browse and edit your data.

4.2 Example Schema

```
model Transaction {
  id             String  @id @default(cuid())
  userId         String
  beneficiaryId String
  sourceAmount  Decimal @db.Decimal(18,2)
  sourceCurrency String
  targetAmount  Decimal @db.Decimal(18,2)
  targetCurrency String
  fxRate        Decimal @db.Decimal(18,6)
  feeFixed      Decimal @db.Decimal(18,2)
  feePct        Decimal @db.Decimal(5,4)
  status        String
  highRisk      Boolean @default(false)
  createdAt     DateTime @default(now())
}
```

4.3 Workflow

1. Define models in `prisma/schema.prisma`.

2. Run `npx prisma migrate dev -n init` to create tables.
3. Use the generated client in your code:

```
const txns = await prisma.transaction.findMany({
  where: { userId },
  orderBy: { createdAt: 'desc' },
  take: 50,
});
```

4.4 Hands-On Exercise

1. Install Prisma: `npm install @prisma/client && npm install -D prisma`.
2. Initialize Prisma: `npx prisma init`.
3. Define a `User` and `Transaction` model in the schema.
4. Run a migration to generate database tables.
5. Write a script to insert a dummy transaction and query it back.

4.5 Mini Project

Build a database-backed service:

- Create a `User` model with fields for id, email, and password.
- Create a `Transaction` model (as above).
- Connect the Express API to Prisma.
- Implement endpoints: `/users` (create user) and `/transactions` (list by user).

This demonstrates how Prisma integrates seamlessly with Express and Postgres.

Chapter 5

Caching: Redis

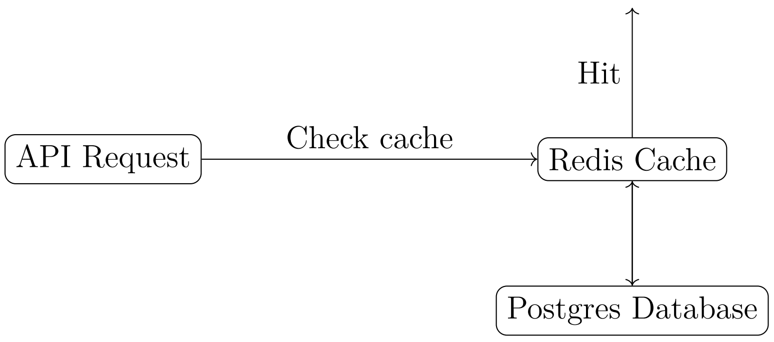
5.1 Concepts

Redis is an open-source, in-memory data store often described as a “key–value database.” Unlike relational databases that write primarily to disk, Redis keeps data in memory, making it extremely fast. It is frequently used for:

- **Caching** – storing API responses, database query results, or computations.
- **Sessions** – managing logged-in user sessions.
- **Rate Limiting** – counting API requests per user/IP.
- **Message Queues** – building producer–consumer pipelines.

Redis supports TTL (time-to-live) values, meaning data can expire after a set duration. This is particularly useful for temporary data like authentication tokens or cached API results.

5.2 Visual Diagram



5.3 Example: Caching Weather by City

For a student dashboard that shows current weather, you can cache responses per city to avoid hitting the weather API for every request:

1. Build a cache key like `weather:city:London`.

2. Use a shorter TTL (e.g., 300 seconds) because weather changes frequently.
3. On a miss, fetch from a public weather API, store it, and serve.

5.4 Code Snippet

```
import Redis from 'ioredis';
const redis = new Redis(process.env.REDIS_URL);

export async function getWeather(city) {
  const key = `weather:city:${city.toLowerCase()}`;
  const cached = await redis.get(key);
  if (cached) return JSON.parse(cached);

  // Replace with a real weather API
  const res = await fetch(`https://api.open-meteo.com/v1/forecast?latitude
     =51.5072&longitude=0.1276&current_weather=true`);
  const data = await res.json();
  await redis.set(key, JSON.stringify(data), 'EX', 300); // 5-minute TTL
  return data;
}
```

5.5 Hands-On Exercise

1. Install Redis locally with Docker: `docker run -p 6379:6379 redis`.
2. Write a Node.js function that caches weather data for a given city.
3. Test the function by requesting the same city twice—verify the second call returns instantly from Redis.
4. Extend the example to handle multiple cities (New York, London, Tokyo) and explore how TTL affects freshness.

Chapter 6

DevOps: Docker and Deployment

6.1 Docker Compose

Now let's wire multiple services together:

```
version: '3.9'
services:
  api:
    build: .
    ports:
      - "8080:8080"
    depends_on:
      - db
      - redis

  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb
    ports:
      - "5432:5432"

  redis:
    image: redis:7
    ports:
      - "6379:6379"
```

With this file, running `docker-compose up -build` spins up API, Postgres, and Redis all at once.

6.2 Deployment Options

Popular beginner-friendly platforms:

- **Railway** – simple, free tier for students, one-click deploys.
- **Render** – great free tier, supports web services and background workers.

- **Neon** – managed Postgres.
- **Upstash** – managed Redis.

Typical steps:

1. Push your code to GitHub.
2. Connect the repo to Railway/Render.
3. Add environment variables (DB connection string, Redis URL, JWT secret).
4. Deploy – the platform builds the Docker image automatically.

6.3 Hands-On Exercise

1. Install Docker Desktop and Docker Compose.
2. Write a Dockerfile for your Express API.
3. Create a docker-compose.yml that wires API + Postgres + Redis.
4. Run locally with `docker-compose up`.
5. Push your project to GitHub and deploy to Railway.

6.4 Mini Project

Take the weather app example from Chapter 5:

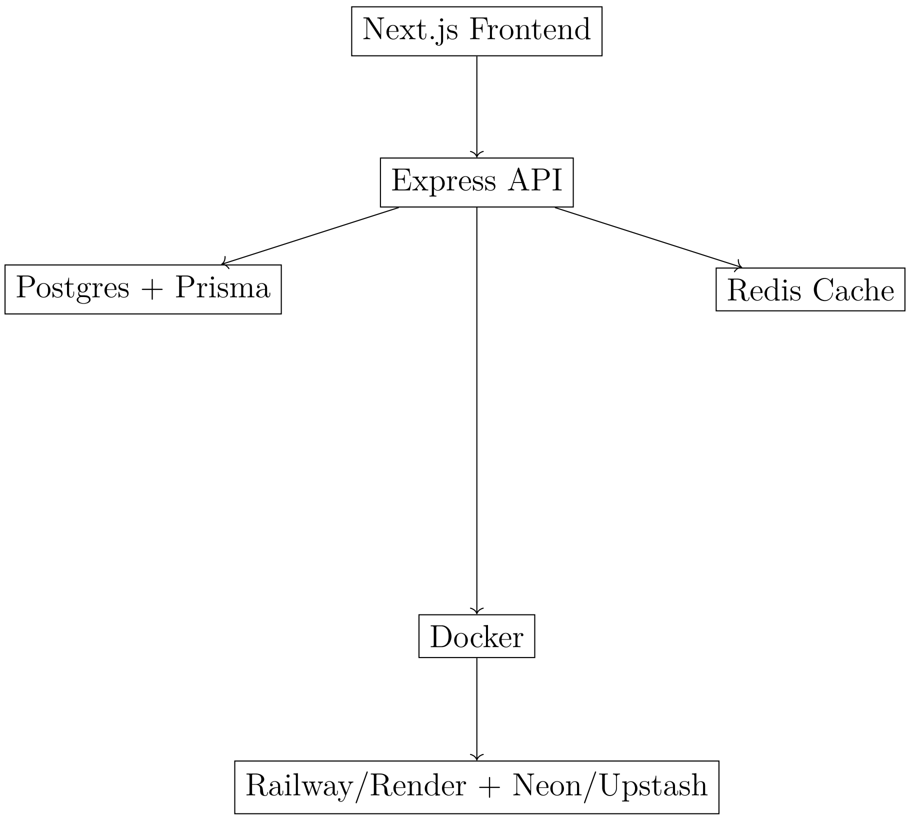
- Containerize it with Docker.
- Use docker-compose to add Redis for caching.
- Deploy to Railway so students can access it online.

This exercise ties together Docker concepts with a practical, student-friendly use case.

Chapter 7

Putting It Together

7.1 Architecture Diagram



Chapter 8

Conclusion

Full stack development is not about memorizing frameworks—it's about understanding how different parts of the system fit together. By working through this book, you've learned how to build applications that are not only functional, but also scalable, reliable, and ready for production.

8.1 Review Points

- Frontend – React helps build UI, while Next.js adds routing, SSR, and API routes.
- Backend – Express provides routing, middleware, and authentication layers.
- Database – PostgreSQL is a robust relational DB, and Prisma simplifies queries with a type-safe client.
- Caching – Redis boosts performance and lowers costs by storing temporary data.
- DevOps – Docker and Docker Compose ensure consistency; cloud services (Railway, Render, Neon, Upstash) make deployment simple.
- Integration – Together, these tools create a modern stack that powers hackathon prototypes and startup MVPs alike.

8.2 Career Applications

Mastering this modern stack prepares you for:

- Hackathons – ship production-ready demos in days, not weeks.
- Internships/Jobs – companies love candidates who can contribute across the stack.
- Startups – launch an MVP fast, iterate quickly, and scale.
- Teaching & Mentoring – guide juniors and peers in full stack concepts.

8.3 Next Steps for Students

Here's how you can keep building:

1. Clone the capstone Weather + Notes App and add new features (file uploads, real-time chat, payments).
2. Join hackathons to apply what you've learned and network with peers.
3. Explore advanced topics: GraphQL, serverless, edge deployments, and AI integrations.
4. Share your projects publicly (GitHub, LinkedIn, Devpost) to build your portfolio.
5. Contribute to open source projects using Prisma, Next.js, or Express.
6. Teach a peer or run a workshop—teaching solidifies your own knowledge.

8.4 Quiz Yourself

Test your knowledge with a few review questions:

1. What is the difference between SSR and SSG in Next.js?
2. Why is Prisma considered type-safe, and how does that help developers?
3. Give two use cases where Redis improves performance.
4. What problem does Docker solve when moving projects between machines?
5. How does Docker Compose simplify running Postgres + Redis + API together?

Final Note: Mastering the modern full stack isn't about learning one tool—it's about learning how to learn, adapt, and integrate. The best developers are problem-solvers who pick the right tool at the right time.

Chapter 9

References

Below are useful resources and official documentation links for further learning:

- React Documentation: `https://react.dev/`
- Next.js Documentation: `https://nextjs.org/docs`
- Express.js Guide: `https://expressjs.com/`
- PostgreSQL Docs: `https://www.postgresql.org/docs/`
- Prisma Documentation: `https://www.prisma.io/docs`
- Redis Documentation: `https://redis.io/docs/`
- Docker Documentation: `https://docs.docker.com/`
- Railway Deployment: `https://docs.railway.app/`
- Render Deployment: `https://render.com/docs`
- Upstash Redis: `https://upstash.com/docs`
- Neon Postgres: `https://neon.tech/docs`