Full-Stack Development

Complete Beginner to Intermediate Notes

Node.js · RESTful API · Express · Redis · React · AWS

Every concept explained with real-world analogies, examples, and expected outputs

Table of Contents

Node.js · RESTful API · Express · Redis · React · AWS

Table of Contents

Node.js · RESTful API · Express · Redis · React · AWS

Chapter 1: Node.js — The Foundation

Imagine a restaurant kitchen. Traditional servers (like Apache) hire a new waiter for every customer — very expensive. Node.js is like one super-efficient waiter who takes everyone's order, shouts it to the kitchen, and immediately takes the next order without waiting around. That's the event loop!

1.1 What is Node.js?

Node.js is a JavaScript runtime environment built on Chrome's V8 JavaScript engine. Before Node.js, JavaScript only ran in browsers. Node.js lets JavaScript run on your computer (server), enabling you to build backend services, APIs, CLIs, and more.

- Released in 2009 by Ryan Dahl
- Uses a single-threaded, non-blocking, event-driven architecture
- Great for I/O-heavy applications (APIs, chat apps, streaming)
- Used by Netflix, LinkedIn, Uber, PayPal, NASA

■ Real World Analogy

Browser JavaScript = A cash register that only works inside a store.
Node.js JavaScript = That same register, now portable — works anywhere: warehouse, office, server.

1.2 Installing Node.js

■ *Step-by-step installation*

```
# Download from https://nodejs.org (choose LTS version)

# Verify installation
node --version
npm --version

# Run a JavaScript file
node myfile.js

# Open interactive REPL (Read-Eval-Print Loop)
node
> 2 + 2
> console.log("Hello")
```

```
$ Output
v20.11.0
10.2.4
```

1.3 Your First Node.js Program

■ *hello.js — Print to terminal*

```
// hello.js
console.log("Hello, World!");
console.log("Today:", new Date().toDateString());

const name = "Alice";
const age = 25;
console.log(`My name is ${name} and I am ${age} years old.`);
```

```
$ Output
Hello, World!
Today: Mon Jan 15 2024
My name is Alice and I am 25 years old.
```

1.4 Node.js Built-in Modules

Node.js ships with built-in modules — no installation needed. Think of them as free tools already in your toolbox.

Module	Purpose	Real-World Use
fs	File system operations	Read/write files, create directories
http	Create HTTP servers	Build a web server from scratch
path	Work with file paths	Join paths safely across OS
os	Operating system info	Get CPU, memory, hostname
events	Event-driven programming	Custom event emitters
crypto	Cryptographic functions	Hash passwords, generate tokens
url	Parse URLs	Extract query strings from URLs
stream	Streaming data	Handle large files efficiently

File System Module (fs)

■ *Real-world example: Reading and writing a config file*

```
const fs = require('fs');

// ■■ SYNCHRONOUS (blocks everything — avoid in production!) ■■
const data = fs.readFileSync('config.json', 'utf8');
console.log('Config loaded:', data);

// ■■ ASYNCHRONOUS with Callback (old style) ■■
fs.readFile('config.json', 'utf8', (err, data) => {
if (err) {
console.error('Could not read file:', err.message);
return;
}
console.log('File contents:', data);
});

// ■■ ASYNC/AWAIT (modern, preferred style) ■■
const fsPromises = require('fs/promises');

async function manageFile() {
// Write a file
await fsPromises.writeFile('notes.txt', 'Node.js is great!', 'utf8');
console.log('File written successfully');

// Read it back
const content = await fsPromises.readFile('notes.txt', 'utf8');
console.log('Read back:', content);

// Append to file
await fsPromises.appendFile('notes.txt', '\nLearning every day.');
console.log('Content appended');

// Check if file exists
try {
await fsPromises.access('notes.txt');
console.log('File exists!');
} catch {
console.log('File does not exist');
}

// Delete the file
await fsPromises.unlink('notes.txt');
console.log('File deleted');
}

manageFile();
```

```
$ Output
File written successfully
Read back: Node.js is great!
Content appended
File exists!
File deleted
```

HTTP Module — Build a Server from Scratch

■ *Real-world example: A simple web server that handles different URLs*

```
const http = require('http');
const url = require('url');

const server = http.createServer((req, res) => {
const parsedUrl = url.parse(req.url, true);
const path = parsedUrl.pathname;
const query = parsedUrl.query;

// Set response headers
res.setHeader('Content-Type', 'application/json');

if (path === '/' && req.method === 'GET') {
res.writeHead(200);
res.end(JSON.stringify({ message: 'Welcome to my server!' }));

} else if (path === '/greet' && req.method === 'GET') {
const name = query.name || 'Stranger';
res.writeHead(200);
res.end(JSON.stringify({ greeting: `Hello, ${name}!` }));

} else {
res.writeHead(404);
res.end(JSON.stringify({ error: 'Not Found' }));
}
});

server.listen(3000, () => {
console.log('Server started on http://localhost:3000');
});
```

```
$ Output
Server started on http://localhost:3000

# Visit http://localhost:3000/
{ "message": "Welcome to my server!" }

# Visit http://localhost:3000/greet?name=Alice
{ "greeting": "Hello, Alice!" }

# Visit http://localhost:3000/unknown
{ "error": "Not Found" }
```

1.5 Asynchronous Programming Deep Dive

This is the most important concept in Node.js. JavaScript is single-threaded, but it can handle many tasks at once using asynchronous patterns.

Callbacks (Old Way — leads to "Callback Hell")

■ *Example: Nested callbacks become hard to read*

```
// Callback Hell Example (avoid this!)
fs.readFile('user.json', (err, userData) => {
if (err) return console.error(err);
const user = JSON.parse(userData);

fs.readFile('orders.json', (err, orderData) => {
if (err) return console.error(err);
const orders = JSON.parse(orderData);

fs.writeFile('report.json', JSON.stringify({ user, orders }), (err) => {
if (err) return console.error(err);
console.log('Report saved!'); // Deeply nested!
});
});
});
```

Promises (Better)

■ *Example: Chaining promises cleanly*

```
const fetchUser = (id) => {
return new Promise((resolve, reject) => {
setTimeout(() => {
if (id > 0) {
resolve({ id, name: 'Alice', email: 'alice@example.com' });
} else {
reject(new Error('Invalid user ID'));
}
}, 1000); // Simulates 1 second DB query
});
};

// Using the promise
fetchUser(1)
.then(user => {
console.log('Got user:', user.name);
return fetchUser(2); // Chain another promise
})
.then(user2 => console.log('Got user2:', user2.name))
.catch(err => console.error('Error:', err.message))
.finally(() => console.log('Done fetching!'));
```

```
$ Output
# After 1 second:
Got user: Alice
# After another 1 second:
Got user2: Alice
Done fetching!
```

Async/Await (Best — Modern Standard)

■ *Real-world example: Fetch user data, then their orders, then save a report*

```
const { readFile, writeFile } = require('fs/promises');

// Simulated database functions
async function getUserFromDB(userId) {
await new Promise(r => setTimeout(r, 500)); // simulate DB delay
return { id: userId, name: 'Alice', email: 'alice@shop.com' };
}

async function getOrdersFromDB(userId) {
await new Promise(r => setTimeout(r, 300));
return [
{ orderId: 101, product: 'Laptop', price: 999 },
{ orderId: 102, product: 'Mouse', price: 29 },
];
}

async function generateReport(userId) {
try {
console.log('Fetching user...');
const user = await getUserFromDB(userId);
console.log(`Got user: ${user.name}`);

console.log('Fetching orders...');
const orders = await getOrdersFromDB(userId);
console.log(`Got ${orders.length} orders`);

const total = orders.reduce((sum, o) => sum + o.price, 0);
const report = { user, orders, totalSpent: total };

await writeFile('report.json', JSON.stringify(report, null, 2));
console.log('Report saved! Total spent:', total);

} catch (error) {
console.error('Failed to generate report:', error.message);
}
}

generateReport(1);
```

```
$ Output
Fetching user...
Got user: Alice
Fetching orders...
Got 2 orders
Report saved! Total spent: 1028
```

■ Async Tip

Use Promise.all() to run multiple async tasks simultaneously:
const [user, orders] = await Promise.all([getUserFromDB(1), getOrdersFromDB(1)]);
This is ~2x faster than awaiting them one by one!

1.6 npm — Node Package Manager

npm is like an app store for Node.js code. Other developers publish "packages" (libraries) that you can download and use in your project.

■ *Essential npm commands explained*

```
npm init -y # Create package.json (project config)
npm install express # Install a package (saves to dependencies)
npm install --save-dev nodemon # Install dev-only package
npm install # Install all packages from package.json
npm uninstall express # Remove a package
npm update # Update all packages
npm run start # Run the 'start' script
npm run dev # Run the 'dev' script
npm list --depth=0 # Show installed packages
npm outdated # Show outdated packages
npx create-react-app my-app # Run a package without installing
```

■ *package.json explained*

```
{
"name": "my-shop-api",
"version": "1.0.0",
"description": "E-commerce REST API",
"main": "server.js",
"scripts": {
"start": "node server.js", // npm start
"dev": "nodemon server.js", // npm run dev (auto-restart on changes)
"test": "jest", // npm test
"build": "tsc" // npm run build
},
"dependencies": {
"express": "^4.18.2", // ^ means any compatible version
"mongoose": "^7.6.3",
"redis": "^4.6.10"
},
"devDependencies": {
"nodemon": "^3.0.1", // Only used during development
"jest": "^29.7.0"
},
"engines": {
"node": ">=18.0.0" // Minimum Node.js version required
}
}
```

1.7 Event Emitter — The Heart of Node.js

■ *Real-world example: Order processing system with events*

```
const EventEmitter = require('events');

class OrderSystem extends EventEmitter {
placeOrder(order) {
console.log(`Order #${order.id} placed for: ${order.product}`);
this.emit('order:placed', order); // Fire an event
}

processPayment(order) {
console.log(`Payment processing for order #${order.id}...`);
this.emit('payment:success', order);
}
}

const system = new OrderSystem();

// Listen for events
system.on('order:placed', (order) => {
console.log(` [Inventory] Reserving stock for: ${order.product}`);
system.processPayment(order);
});

system.on('payment:success', (order) => {
console.log(` [Email] Sending confirmation to ${order.email}`);
console.log(` [Shipping] Creating shipment for order #${order.id}`);
});

// Trigger the flow
system.placeOrder({ id: 1001, product: 'MacBook Pro', email: 'alice@example.com' });
```

```
$ Output
Order #1001 placed for: MacBook Pro
[Inventory] Reserving stock for: MacBook Pro
Payment processing for order #1001...
[Email] Sending confirmation to alice@example.com
[Shipping] Creating shipment for order #1001
```

Chapter 2: RESTful API — How the Web Communicates

REST (Representational State Transfer) is a set of rules for building web APIs. Think of it like ordering food at a restaurant: you (client) give your order (request) to the waiter (API), who brings it from the kitchen (server) back to you (response).

2.1 What is an API?

API = Application Programming Interface. It's a way for two software programs to talk to each other. When you check the weather on your phone, it's using a weather API to get data from a server.

■ Real World Analogy
Restaurant Menu = API Documentation (tells you what's available)
Your order = HTTP Request (what you're asking for)
Waiter = API (carries your request to the kitchen)
Kitchen = Server/Database (processes and prepares the response)
Your food = HTTP Response (what you receive back)

2.2 HTTP Methods (CRUD Operations)

Every action you do on the web maps to an HTTP method. CRUD stands for Create, Read, Update, Delete.

HTTP Method	CRUD Action	Real Example	Success Code
GET	Read	Load your Twitter feed	200 OK
POST	Create	Post a new tweet	201 Created
PUT	Update (Full)	Replace your entire profile	200 OK
PATCH	Update (Part)	Change just your username	200 OK
DELETE	Delete	Delete a tweet	204 No Content

2.3 HTTP Status Codes — What They Mean

Code	Name	Meaning	When It Happens
200	OK	Everything worked	Successful GET or PUT
201	Created	New resource was made	Successful POST
204	No Content	Success, nothing to return	Successful DELETE
400	Bad Request	You sent bad data	Missing required field
401	Unauthorized	You need to log in	No auth token sent
403	Forbidden	You don't have permission	Accessing someone else's data
404	Not Found	Resource doesn't exist	User ID doesn't exist
409	Conflict	Data conflict	Email already registered
422	Unprocessable	Validation failed	Email format is wrong
429	Too Many Requests	Rate limited	API limit exceeded
500	Internal Server Err	Something broke server-side	Unhandled exception in code
503	Service Unavailable	Server is down/overloaded	Maintenance or crash

2.4 Anatomy of an HTTP Request & Response

■ *Full request/response cycle explained*

```
# ■■ HTTP REQUEST ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
POST /api/users HTTP/1.1
Host: api.example.com
Content-Type: application/json ← Tells server what format we're sending
Authorization: Bearer eyJhbGci... ← Auth token
Accept: application/json ← Tells server what format we want back

{ ← Request Body (the data we're sending)
"name": "Alice Johnson",
"email": "alice@example.com",
"password": "secret123"
}

# ■■ HTTP RESPONSE ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/users/42 ← Where the new resource lives

{ ← Response Body
"id": 42,
"name": "Alice Johnson",
"email": "alice@example.com",
"createdAt": "2024-01-15T10:30:00Z"
// Note: password is NOT returned!
}
```

2.5 REST API Design Best Practices

URL Design Rules

■ *Good vs Bad URL patterns*

```
# ■ GOOD — Use nouns, not verbs
GET /api/users # Get all users
GET /api/users/42 # Get user with ID 42
POST /api/users # Create a user
PUT /api/users/42 # Update user 42
DELETE /api/users/42 # Delete user 42
GET /api/users/42/posts # Get posts belonging to user 42
GET /api/products?category=electronics&sort=price # Filtering

# ■ BAD — Do not use verbs in URLs
GET /getUsers
POST /createNewUser
GET /deleteUser?id=42
POST /updateUserProfile
```

API Versioning

■ *Always version your API to avoid breaking changes*

```
# URL versioning (most common)
https://api.example.com/v1/users
https://api.example.com/v2/users ← New version, old still works

# Header versioning
GET /api/users
Accept: application/vnd.example.v2+json

// Express setup
app.use('/api/v1', require('./routes/v1/users'));
app.use('/api/v2', require('./routes/v2/users'));
```

Consistent JSON Response Structure

■ *Always use a predictable response format*

```
// ■ Consistent success response
{
"success": true,
"data": {
"id": 42,
"name": "Alice"
},
"meta": {
"page": 1,
"perPage": 10,
"total": 150
}
}

// ■ Consistent error response
{
"success": false,
"error": {
"code": "VALIDATION_ERROR",
"message": "Email is required",
"field": "email"
}
}
```

2.6 Authentication: JWT Tokens

JWT = JSON Web Token. When a user logs in, the server creates a token (like a signed wristband at a concert). The user sends this token with every future request to prove who they are.

■ *How JWT works step by step*

```
// STEP 1: User logs in
POST /api/auth/login
Body: { "email": "alice@example.com", "password": "secret123" }

// STEP 2: Server verifies credentials and returns token
Response: {
"token": "eyJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOjQyfQ.abc123"
}

// STEP 3: User sends token with every subsequent request
GET /api/profile
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOjQyfQ.abc123

// STEP 4: Server decodes the token to identify user
// Token structure: header.payload.signature
// Decoded payload: { "userId": 42, "email": "alice@example.com", "exp": 1705320000 }
```

■ *JWT implementation with Node.js*

```
npm install jsonwebtoken bcryptjs

const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const SECRET = process.env.JWT_SECRET; // 'mysupersecretkey'

// Hash password before storing
async function hashPassword(plainText) {
const salt = await bcrypt.genSalt(10);
const hashed = await bcrypt.hash(plainText, salt);
return hashed; // '$2a$10$...' (cannot be reversed)
}

// Login endpoint
app.post('/api/login', async (req, res) => {
const { email, password } = req.body;

const user = await User.findOne({ email });
if (!user) return res.status(401).json({ error: 'Invalid credentials' });

const isValid = await bcrypt.compare(password, user.passwordHash);
if (!isValid) return res.status(401).json({ error: 'Invalid credentials' });

// Create token valid for 7 days
const token = jwt.sign(
{ userId: user.id, email: user.email, role: user.role },
SECRET,
{ expiresIn: '7d' }
);

res.json({ token, user: { id: user.id, name: user.name } });
});

// Middleware to protect routes
function authenticate(req, res, next) {
const authHeader = req.headers['authorization'];
const token = authHeader && authHeader.split(' ')[1]; // 'Bearer TOKEN'

if (!token) return res.status(401).json({ error: 'No token provided' });

try {
const decoded = jwt.verify(token, SECRET);
req.user = decoded; // { userId, email, role }
next();
} catch (err) {
res.status(403).json({ error: 'Invalid or expired token' });
}
}

// Protected route
app.get('/api/profile', authenticate, (req, res) => {
res.json({ message: `Hello, ${req.user.email}!` });
});
```

```
$ Output
POST /api/login (valid credentials)
→ 200 { "token": "eyJhbGci...", "user": { "id": 1, "name": "Alice" } }

GET /api/profile (with valid token)
→ 200 { "message": "Hello, alice@example.com!" }

GET /api/profile (no token)
→ 401 { "error": "No token provided" }
```

1.4 – 1.7 Built-in Module Reference

The modules below are all built into Node.js — no npm install required.
Just require() them directly in your code.

Module	How to import	npm install needed?
`fs`	`require('fs')` / `require('fs/promises')`	No — built-in
`http`	`require('http')`	No — built-in
`path`	`require('path')`	No — built-in
`os`	`require('os')`	No — built-in
`events`	`require('events')`	No — built-in
`crypto`	`require('crypto')`	No — built-in
`url`	`require('url')`	No — built-in
`stream`	`require('stream')`	No — built-in

Third-Party Packages → Need npm install

Unlike built-in modules, packages from npm must be installed before use.

```
npm install express
npm install jsonwebtoken bcryptjs
npm install redis
npm install @aws-sdk/client-s3
npm install express-validator
```

> **Quick Rule**
>
> If you see `require('module')` with no 'npm install' shown nearby, it's a built-in Node.js module — no install step needed.
>
> If the chapter shows `npm install <package>` first, you must run that command in your project folder before the require() will work.

1.7 EventEmitter — No Install Required

The EventEmitter example on the previous chapter uses require('events'), which is built-in. The class-based pattern shown is the idiomatic way to implement event-driven systems without any extra dependencies.

```
$ Output
# Verify: node -e "require('events'); console.log('OK')"
OK    ← no install step needed
```

Chapter 3: Express.js — Building Web Applications

Express is a minimal, unopinionated web framework for Node.js. It handles the boring parts of HTTP so you can focus on your application logic. Think of it as the scaffolding of a building — it gives structure without dictating what goes inside.

3.1 Setting Up Express

■ *Step-by-step project setup*

```
# Create project
mkdir my-api && cd my-api
npm init -y
npm install express
npm install --save-dev nodemon

# Add to package.json scripts:
"scripts": {
"start": "node server.js",
"dev": "nodemon server.js"
}
```

■ *server.js — Complete Express boilerplate*

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

// ■■ BUILT-IN MIDDLEWARE ■■■■■■■■■■■■■■■■■■■■
app.use(express.json()); // Parse JSON request bodies
app.use(express.urlencoded({ extended: true })); // Parse form data
app.use(express.static('public')); // Serve static files from /public folder

// ■■ ROUTES ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
app.get('/', (req, res) => {
res.json({ message: 'API is running!', version: '1.0' });
});

// ■■ START SERVER ■■■■■■■■■■■■■■■■■■■■■■■■■
app.listen(PORT, () => {
console.log(`Server running on http://localhost:${PORT}`);
});

module.exports = app; // Export for testing
```

```
$ Output
Server running on http://localhost:3000
# GET / → { "message": "API is running!", "version": "1.0" }
```

3.2 Request Object (req) — What You Receive

■ *All the ways to get data from a request*

```
app.get('/api/users/:id', (req, res) => {
// URL parameters (/users/42 → id = '42')
const userId = req.params.id;

// Query string (?page=2&limit=10)
const page = req.query.page || 1;
const limit = req.query.limit || 10;

// Request headers
const token = req.headers['authorization'];
const contentType = req.headers['content-type'];

// Request body (POST/PUT)
const { name, email } = req.body;

// Request method (GET, POST, etc.)
console.log('Method:', req.method);

// Full URL
console.log('URL:', req.originalUrl);

// Client IP
console.log('IP:', req.ip);

res.json({ userId, page, limit });
});

// GET /api/users/42?page=2&limit=5
```

```
$ Output
Method: GET
URL: /api/users/42?page=2&limit=5
IP: ::1
→ { "userId": "42", "page": "2", "limit": "5" }
```

3.3 Response Object (res) — What You Send

■ *All response methods*

```
// Send JSON
res.json({ name: 'Alice' });

// Set status code then send JSON
res.status(201).json({ id: 42, message: 'Created' });

// Send plain text
res.send('Hello World');

// Send HTML
res.send('<h1>Hello World</h1>');

// Redirect to another URL
res.redirect('/api/v2/users');
res.redirect(301, 'https://newdomain.com'); // Permanent redirect

// Set response headers
res.setHeader('X-Custom-Header', 'MyValue');
res.setHeader('Cache-Control', 'max-age=3600');

// Download a file
res.download('/path/to/file.pdf', 'report.pdf');

// Send 204 No Content (e.g., after DELETE)
res.status(204).send();

// Send 404
res.status(404).json({ error: 'User not found' });
```

3.4 Middleware — The Power of Express

Middleware are functions that run BETWEEN the request arriving and the response being sent. Every middleware has access to req, res, and next(). Calling next() passes control to the next middleware.

■ *Real-world: E-commerce API middleware stack*

```
const express = require('express');
const app = express();

// ■■ 1. Request Logger ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
const requestLogger = (req, res, next) => {
const start = Date.now();

// Run after response is sent
res.on('finish', () => {
const duration = Date.now() - start;
console.log(`[${new Date().toISOString()}] ${req.method} ${req.url} → ${res.statusCode}
(${duration}ms)`);
});

next(); // IMPORTANT: pass to next middleware
};

// ■■ 2. CORS Middleware ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
const cors = (req, res, next) => {
res.setHeader('Access-Control-Allow-Origin', '*');
res.setHeader('Access-Control-Allow-Methods', 'GET,POST,PUT,PATCH,DELETE');
res.setHeader('Access-Control-Allow-Headers', 'Content-Type,Authorization');

if (req.method === 'OPTIONS') return res.status(200).end(); // Preflight
next();
};

// ■■ 3. Rate Limiter ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
const requestCounts = {};
const rateLimiter = (req, res, next) => {
const ip = req.ip;
const now = Date.now();

if (!requestCounts[ip]) requestCounts[ip] = [];
requestCounts[ip] = requestCounts[ip].filter(t => now - t < 60000); // 1 min window
requestCounts[ip].push(now);

if (requestCounts[ip].length > 100) {
return res.status(429).json({ error: 'Too many requests. Wait 1 minute.' });
}
next();
};

// ■■ 4. JWT Auth Middleware ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
const authenticate = (req, res, next) => {
const token = req.headers['authorization']?.split(' ')[1];
if (!token) return res.status(401).json({ error: 'Authentication required' });

try {
req.user = jwt.verify(token, process.env.JWT_SECRET);
next();
} catch {
res.status(403).json({ error: 'Invalid or expired token' });
}
};
```

```
// ■■ 5. Admin Only ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
const adminOnly = (req, res, next) => {
if (req.user.role !== 'admin') {
return res.status(403).json({ error: 'Admin access required' });
}
next();
};

// Apply middleware globally
app.use(requestLogger);
app.use(cors);
app.use(rateLimiter);
app.use(express.json());

// Public routes (no auth needed)
app.get('/api/products', getProducts);

// Protected routes (auth required)
app.get('/api/profile', authenticate, getProfile);
app.post('/api/orders', authenticate, createOrder);

// Admin-only routes
app.delete('/api/users/:id', authenticate, adminOnly, deleteUser);
```

```
$ Output
[2024-01-15T10:30:00.000Z] GET /api/products → 200 (12ms)
[2024-01-15T10:30:05.000Z] POST /api/orders → 201 (45ms)
[2024-01-15T10:30:06.000Z] DELETE /api/users/5 → 403 (3ms)
```

3.5 Express Router — Organizing Routes

■ *Project structure for a real e-commerce API*

```
■ my-api/
■ routes/
■ auth.js ← /api/auth/login, /api/auth/register
■ users.js ← /api/users (CRUD)
■ products.js ← /api/products (CRUD)
■ orders.js ← /api/orders (CRUD)
■ middleware/
■ auth.js ← JWT authentication
■ validate.js ← Input validation
■ controllers/
■ userController.js
■ productController.js
■ server.js ← Main entry point
■ package.json
■ .env
```

■ *routes/products.js — Full product routes*

```
const express = require('express');
const router = express.Router();
const { authenticate, adminOnly } = require('../middleware/auth');

// In-memory data (use a real DB in production)
let products = [
{ id: 1, name: 'Laptop', price: 999, category: 'electronics', stock: 50 },
{ id: 2, name: 'Headphones', price: 199, category: 'electronics', stock: 200 },
{ id: 3, name: 'Desk', price: 349, category: 'furniture', stock: 30 },
];
let nextId = 4;

// GET /api/products?category=electronics&sort=price&page=1&limit=10
router.get('/', (req, res) => {
let result = [...products];

// Filter by category
if (req.query.category) {
result = result.filter(p => p.category === req.query.category);
}

// Search by name
if (req.query.search) {
const q = req.query.search.toLowerCase();
result = result.filter(p => p.name.toLowerCase().includes(q));
}

// Sort
if (req.query.sort === 'price') {
result.sort((a, b) => a.price - b.price);
}

// Pagination
const page = parseInt(req.query.page) || 1;
const limit = parseInt(req.query.limit) || 10;
const start = (page - 1) * limit;
const paginated = result.slice(start, start + limit);

res.json({
success: true,
data: paginated,
meta: { page, limit, total: result.length }
});
});

// GET /api/products/:id
router.get('/:id', (req, res) => {
const product = products.find(p => p.id === parseInt(req.params.id));
if (!product) return res.status(404).json({ success: false, error: 'Product not found' });
res.json({ success: true, data: product });
});

// POST /api/products (admin only)
router.post('/', authenticate, adminOnly, (req, res) => {
const { name, price, category, stock } = req.body;
if (!name || !price) return res.status(400).json({ error: 'Name and price are required' });

const product = { id: nextId++, name, price, category, stock: stock || 0 };
products.push(product);
res.status(201).json({ success: true, data: product });
});

// PATCH /api/products/:id (admin only)
router.patch('/:id', authenticate, adminOnly, (req, res) => {
```

```
const idx = products.findIndex(p => p.id === parseInt(req.params.id));
if (idx === -1) return res.status(404).json({ error: 'Product not found' });

products[idx] = { ...products[idx], ...req.body, id: products[idx].id };
res.json({ success: true, data: products[idx] });
});

// DELETE /api/products/:id (admin only)
router.delete('/:id', authenticate, adminOnly, (req, res) => {
const idx = products.findIndex(p => p.id === parseInt(req.params.id));
if (idx === -1) return res.status(404).json({ error: 'Product not found' });

products.splice(idx, 1);
res.status(204).send();
});

module.exports = router;
```

```
$ Output
GET /api/products?category=electronics&sort=price
→ 200 {
"success": true,
"data": [
{ "id": 2, "name": "Headphones", "price": 199, ... },
{ "id": 1, "name": "Laptop", "price": 999, ... }
],
"meta": { "page": 1, "limit": 10, "total": 2 }
}

POST /api/products (without admin token)
→ 401 { "error": "Authentication required" }
```

■ *server.js — Mounting all routes*

```
const express = require('express');
const app = express();

app.use(express.json());

// Mount routes
app.use('/api/auth', require('./routes/auth'));
app.use('/api/users', require('./routes/users'));
app.use('/api/products', require('./routes/products'));
app.use('/api/orders', require('./routes/orders'));

// 404 handler (catch-all)
app.use((req, res) => {
res.status(404).json({ error: `Route ${req.method} ${req.url} not found` });
});

// Global error handler (must have 4 params!)
app.use((err, req, res, next) => {
console.error('Unhandled error:', err.stack);
res.status(err.status || 500).json({
success: false,
error: err.message || 'Internal Server Error'
});
});

app.listen(3000, () => console.log('API ready on port 3000'));
```

3.6 Input Validation

■ *Using express-validator to validate inputs*

```
npm install express-validator

const { body, validationResult } = require('express-validator');

// Validation rules
const createUserRules = [
body('name').trim().notEmpty().withMessage('Name is required'),
body('email').isEmail().normalizeEmail().withMessage('Valid email required'),
body('password').isLength({ min: 8 }).withMessage('Password min 8 chars'),
body('age').optional().isInt({ min: 13, max: 120 }).withMessage('Age must be 13-120'),
];

// Validation middleware
const validate = (req, res, next) => {
const errors = validationResult(req);
if (!errors.isEmpty()) {
return res.status(422).json({
success: false,
errors: errors.array().map(e => ({ field: e.path, message: e.msg }))
});
}
next();
};

// Apply to route
app.post('/api/users', createUserRules, validate, async (req, res) => {
const { name, email, password } = req.body;
// At this point, data is guaranteed to be valid
const user = await User.create({ name, email, password });
res.status(201).json({ success: true, data: user });
});
```

```
$ Output
POST /api/users { "name": "", "email": "invalid", "password": "abc" }
→ 422 {
"success": false,
"errors": [
{ "field": "name", "message": "Name is required" },
{ "field": "email", "message": "Valid email required" },
{ "field": "password", "message": "Password min 8 chars" }
]
}
```

Chapter 4: Redis — In-Memory Data Store

Redis is like a super-fast sticky note board. Your normal database (like PostgreSQL) is stored on a hard drive and takes time to read. Redis stores everything in RAM (memory), making reads/writes 100-1000x faster.

> ■ **Real World Analogy**
> Database (PostgreSQL) = A library with millions of books. Organized, permanent, but slow to find things.
> Redis = A whiteboard right next to you. You write the most-used info on it for instant access.
> When Redis doesn't have something (cache miss), you fetch it from the library and write it on the whiteboard.

4.1 Installing and Connecting to Redis

■ *Install Redis server and Node.js client*

```
# Install Redis server (Ubuntu/Mac)
sudo apt install redis-server # Ubuntu
brew install redis # macOS

# Start Redis
redis-server

# Test with Redis CLI
redis-cli ping
# → PONG

# Install Node.js client
npm install redis
```

■ *Connection setup with error handling*

```
const redis = require('redis');

const client = redis.createClient({
url: process.env.REDIS_URL || 'redis://localhost:6379',
socket: {
connectTimeout: 5000, // 5 seconds
reconnectStrategy: (retries) => {
if (retries > 5) return new Error('Max retries reached');
return Math.min(retries * 100, 3000); // Exponential backoff
}
}
});

client.on('connect', () => console.log('Redis connected'));
client.on('ready', () => console.log('Redis ready'));
client.on('error', (e) => console.error('Redis error:', e.message));
client.on('reconnecting', () => console.log('Redis reconnecting...'));

await client.connect();
```

```
$ Output
Redis connected
Redis ready
```

4.2 Core Commands — Strings

■ *The most basic Redis data type: key-value strings*

```
// SET and GET
await client.set('username', 'alice');
const val = await client.get('username');
console.log(val); // 'alice'

// SET with expiry (TTL = Time To Live)
await client.setEx('otp:alice', 300, '483921'); // Expires in 5 minutes
const otp = await client.get('otp:alice');
console.log(otp); // '483921'
const ttl = await client.ttl('otp:alice');
console.log(`Expires in: ${ttl}s`); // 'Expires in: 299s'

// DELETE
await client.del('username');

// Check if key exists
const exists = await client.exists('username');
console.log(exists); // 0 (false) or 1 (true)

// Increment / Decrement (for counters)
await client.set('pageViews', 0);
await client.incr('pageViews'); // 1
await client.incr('pageViews'); // 2
await client.incrBy('pageViews', 5); // 7
const views = await client.get('pageViews');
console.log('Views:', views); // 7

// GET multiple keys at once
await client.mSet({ 'color': 'blue', 'size': 'large', 'brand': 'Nike' });
const [color, size] = await client.mGet(['color', 'size']);
console.log(color, size); // 'blue' 'large'
```

```
$ Output
alice
483921
Expires in: 299s
0
Views: 7
blue large
```

4.3 Hashes — Storing Objects

■ *Real-world: Store user session data*

```
// Store a user object as a hash
await client.hSet('user:1001', {
name: 'Alice Johnson',
email: 'alice@example.com',
role: 'admin',
loginAt: new Date().toISOString()
});

// Get a single field
const name = await client.hGet('user:1001', 'name');
console.log('Name:', name);

// Get all fields
const user = await client.hGetAll('user:1001');
console.log('User:', user);

// Update a single field
await client.hSet('user:1001', 'role', 'superadmin');

// Delete a field
await client.hDel('user:1001', 'loginAt');

// Check if field exists
const hasEmail = await client.hExists('user:1001', 'email');
console.log('Has email:', hasEmail);

// Increment a numeric field
await client.hSet('user:1001', 'loginCount', 0);
await client.hIncrBy('user:1001', 'loginCount', 1);
console.log('Login count:', await client.hGet('user:1001', 'loginCount'));
```

```
$ Output
Name: Alice Johnson
User: { name: 'Alice Johnson', email: 'alice@example.com', role: 'admin', loginAt: '2024-01-15...' }
Has email: true
Login count: 1
```

4.4 Lists — Queues and Stacks

■ *Real-world: Task queue for background jobs*

```
// LISTS can be used as queues (FIFO) or stacks (LIFO)

// ■■ QUEUE (First In, First Out) ■■■■■■■■■■■■■■■■■■■■■■■■■■■
// Push to right (enqueue)
await client.rPush('job:queue', JSON.stringify({ type: 'email', to: 'alice@example.com' }));
await client.rPush('job:queue', JSON.stringify({ type: 'resize', file: 'photo.jpg' }));
await client.rPush('job:queue', JSON.stringify({ type: 'email', to: 'bob@example.com' }));

console.log('Queue length:', await client.lLen('job:queue'));

// Pop from left (dequeue) — processes oldest first
const rawJob = await client.lPop('job:queue');
const job = JSON.parse(rawJob);
console.log('Processing job:', job);

// BLPOP — Block until a job is available (for workers)
const result = await client.blPop('job:queue', 30); // Wait up to 30 seconds
if (result) {
const { key, element } = result;
console.log('Got job from:', key, JSON.parse(element));
}

// ■■ RECENT ACTIVITY LOG ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
// Keep last 10 activities
await client.lPush('activity:user:1', 'Logged in');
await client.lPush('activity:user:1', 'Viewed product #42');
await client.lPush('activity:user:1', 'Added to cart');
await client.lTrim('activity:user:1', 0, 9); // Keep only 10 items

const activities = await client.lRange('activity:user:1', 0, -1);
console.log('Recent activity:', activities);
```

```
$ Output
Queue length: 3
Processing job: { type: 'email', to: 'alice@example.com' }
Got job from: job:queue { type: 'resize', file: 'photo.jpg' }
Recent activity: ['Added to cart', 'Viewed product #42', 'Logged in']
```

4.5 Sets — Unique Collections

■ *Real-world: Tracking unique visitors and tags*

```
// SETS store unique values — duplicates automatically ignored

// Track unique visitors to a product page
await client.sAdd('product:42:visitors', 'user:1001');
await client.sAdd('product:42:visitors', 'user:1002');
await client.sAdd('product:42:visitors', 'user:1001'); // Duplicate — ignored

const uniqueVisitors = await client.sCard('product:42:visitors');
console.log('Unique visitors:', uniqueVisitors); // 2

// Check if a user has visited
const hasVisited = await client.sIsMember('product:42:visitors', 'user:1001');
console.log('Has user 1001 visited?', hasVisited); // true

// Get all members
const visitors = await client.sMembers('product:42:visitors');
console.log('All visitors:', visitors);

// ■■ SET OPERATIONS ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
await client.sAdd('user:1001:interests', 'nodejs', 'react', 'redis');
await client.sAdd('user:1002:interests', 'react', 'python', 'nodejs');

// Intersection: What do both users like?
const common = await client.sInter(['user:1001:interests', 'user:1002:interests']);
console.log('Common interests:', common); // ['nodejs', 'react']

// Union: All interests combined
const all = await client.sUnion(['user:1001:interests', 'user:1002:interests']);
console.log('All interests:', all);

// Difference: What user 1001 likes that user 1002 doesn't
const unique1001 = await client.sDiff(['user:1001:interests', 'user:1002:interests']);
console.log('Unique to user 1001:', unique1001); // ['redis']
```

```
$ Output
Unique visitors: 2
Has user 1001 visited? true
All visitors: ['user:1001', 'user:1002']
Common interests: ['nodejs', 'react']
All interests: ['nodejs', 'react', 'redis', 'python']
Unique to user 1001: ['redis']
```

4.6 Sorted Sets — Leaderboards & Rankings

■ *Real-world: Game leaderboard with scores*

```
// Sorted sets: like sets but each member has a SCORE

// Add players with scores
await client.zAdd('game:leaderboard', [
{ score: 15420, value: 'player:alice' },
{ score: 23100, value: 'player:bob' },
{ score: 9800, value: 'player:carol' },
{ score: 31500, value: 'player:dave' },
{ score: 19200, value: 'player:eve' },
]);

// Get top 3 players (highest scores first)
const top3 = await client.zRangeWithScores('game:leaderboard', 0, 2, { REV: true });
top3.forEach((p, i) => {
console.log(`#${i+1}: ${p.value.split(':')[1]} — ${p.score.toLocaleString()} pts`);
});

// Get Alice's rank (0-indexed, from highest)
const rank = await client.zRevRank('game:leaderboard', 'player:alice');
console.log(`Alice's rank: #${rank + 1}`);

// Get Alice's score
const score = await client.zScore('game:leaderboard', 'player:alice');
console.log(`Alice's score: ${score}`);

// Update Alice's score (add 5000 points)
await client.zIncrBy('game:leaderboard', 5000, 'player:alice');
const newScore = await client.zScore('game:leaderboard', 'player:alice');
console.log(`Alice's new score: ${newScore}`);
```

```
$ Output
#1: dave — 31,500 pts
#2: bob — 23,100 pts
#3: eve — 19,200 pts
Alice's rank: #4
Alice's score: 15420
Alice's new score: 20420
```

4.7 Redis as a Caching Layer

■ *Real-world: Cache expensive database queries in Express*

```
// Caching middleware
const cacheMiddleware = (keyFn, ttlSeconds = 300) => async (req, res, next) => {
const cacheKey = keyFn(req);

try {
const cached = await redisClient.get(cacheKey);
if (cached) {
console.log(`CACHE HIT: ${cacheKey}`);
return res.json(JSON.parse(cached));
}
console.log(`CACHE MISS: ${cacheKey}`);

// Override res.json to intercept and cache the response
const originalJson = res.json.bind(res);
res.json = async (data) => {
await redisClient.setEx(cacheKey, ttlSeconds, JSON.stringify(data));
return originalJson(data);
};
next();

} catch (err) {
console.error('Cache error:', err.message);
next(); // Continue without cache on error
}
};

// Cache invalidation
async function invalidateProductCache(productId) {
await redisClient.del(`product:${productId}`);
await redisClient.del('products:all');
console.log('Product cache cleared');
}

// Use caching
app.get('/api/products',
cacheMiddleware(() => 'products:all', 600), // Cache for 10 mins
async (req, res) => {
const products = await Product.findAll(); // Slow DB query
res.json({ data: products });
}
);

app.get('/api/products/:id',
cacheMiddleware(req => `product:${req.params.id}`, 600),
async (req, res) => {
const product = await Product.findById(req.params.id);
res.json({ data: product });
}
);

// Clear cache when product is updated
app.put('/api/products/:id', authenticate, adminOnly, async (req, res) => {
const product = await Product.updateById(req.params.id, req.body);
await invalidateProductCache(req.params.id);
res.json({ data: product });
});
```

```
$ Output
# First request (cold):
```

```
CACHE MISS: products:all
# → Queries database: 234ms

# Subsequent requests (warm):
CACHE HIT: products:all
# → Served from Redis: 2ms ← 117x faster!

# After updating a product:
Product cache cleared
# Next request fetches fresh from DB
```

Chapter 5: React — Building User Interfaces

React is a JavaScript library for building user interfaces. Instead of manually updating the webpage when data changes, React automatically re-renders only the parts that changed. Think of it like a smart whiteboard that erases and redraws only the parts you changed — not the whole board.

5.1 Core Concepts

- Components: Reusable, self-contained pieces of UI (like LEGO bricks)
- JSX: HTML-like syntax written inside JavaScript
- State: Data that belongs to a component and changes over time
- Props: Data passed from parent to child components (like function arguments)
- Virtual DOM: React's efficient way of updating the real browser DOM
- Hooks: Functions that let you use state and other React features in functional components

5.2 JSX — JavaScript + HTML

■ *JSX rules every beginner must know*

```
// JSX looks like HTML but it's JavaScript!

// ■ JSX rules:
// 1. Use className instead of class
<div className='container'>...</div>

// 2. Self-close single elements
<img src='photo.jpg' alt='A photo' />
<input type='text' />
<br />

// 3. Embed JavaScript with {}
const name = 'Alice';
const age = 25;
<p>Hello, {name}! You are {age} years old.</p>
<p>Next year you'll be {age + 1}.</p>

// 4. Conditional rendering
const isLoggedIn = true;
<div>{isLoggedIn ? <Dashboard /> : <LoginPage />}</div>
<div>{isLoggedIn && <WelcomeBanner />}</div>

// 5. Wrap multiple elements in one parent
// ■ BAD
return (<h1>Hello</h1><p>World</p>) // Error!

// ■ GOOD - Use a Fragment
return (
<>
<h1>Hello</h1>
<p>World</p>
</>
);

// 6. Lists must have a key prop
const items = ['Apple', 'Banana', 'Cherry'];
<ul>
{items.map((item, index) => (
<li key={index}>{item}</li>
))}
</ul>
```

5.3 Props — Passing Data to Components

■ *Real-world: Product card component with props*

```
// ProductCard.jsx — reusable component
function ProductCard({ name, price, image, rating, onAddToCart }) {
return (
<div className='card'>
<img src={image} alt={name} />
<h2>{name}</h2>
<p className='price'>${price.toFixed(2)}</p>
<div className='rating'>
{'■'.repeat(Math.round(rating))} ({rating}/5)
</div>
<button onClick={() => onAddToCart(name)}>
Add to Cart
</button>
</div>
);
}

// Default props — used when prop is not provided
ProductCard.defaultProps = {
rating: 0,
image: '/placeholder.jpg'
};

// App.jsx — using ProductCard
function App() {
const handleAddToCart = (productName) => {
alert(`${productName} added to cart!`);
};

const products = [
{ id: 1, name: 'Laptop', price: 999.99, rating: 4.5 },
{ id: 2, name: 'Headphones', price: 199.99, rating: 4.8 },
{ id: 3, name: 'Keyboard', price: 79.99, rating: 4.2 },
];

return (
<div className='product-grid'>
{products.map(product => (
<ProductCard
key={product.id}
name={product.name}
price={product.price}
rating={product.rating}
onAddToCart={handleAddToCart}
/>
))}
</div>
);
}
```

```
$ Output
Renders 3 product cards in a grid layout.
Clicking 'Add to Cart' on 'Laptop' → alert: 'Laptop added to cart!'
```

5.4 useState — Managing Component State

■ *Real-world: Shopping cart with useState*

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

function ShoppingCart() {
const [cart, setCart] = useState([]); // Array of items
const [coupon, setCoupon] = useState(''); // String
const [discount,setDiscount]= useState(0); // Number
const [checkout,setCheckout]= useState(false); // Boolean

const products = [
{ id: 1, name: 'Laptop', price: 999 },
{ id: 2, name: 'Mouse', price: 29 },
];

const addToCart = (product) => {
setCart(prevCart => {
const exists = prevCart.find(item => item.id === product.id);
if (exists) {
// Update quantity
return prevCart.map(item =>
item.id === product.id
? { ...item, qty: item.qty + 1 }
: item
);
}
return [...prevCart, { ...product, qty: 1 }];
});
};

const removeFromCart = (productId) => {
setCart(cart.filter(item => item.id !== productId));
};

const applyCoupon = () => {
if (coupon === 'SAVE20') {
setDiscount(20);
alert('20% discount applied!');
} else {
alert('Invalid coupon');
}
};

const subtotal = cart.reduce((sum, item) => sum + item.price * item.qty, 0);
const total = subtotal * (1 - discount / 100);

if (checkout) {
return <h2>■ Order placed! Total: ${total.toFixed(2)}</h2>;
}

return (
<div>
<h1>Shop</h1>
{products.map(p => (
<button key={p.id} onClick={() => addToCart(p)}>
Add {p.name} (${p.price})
</button>
))}

<h2>Cart ({cart.length} items)</h2>
{cart.map(item => (
```

```
<div key={item.id}>
{item.name} x{item.qty} = ${item.price * item.qty}
<button onClick={() => removeFromCart(item.id)}>Remove</button>
</div>
))}

<p>Subtotal: ${subtotal}</p>
<input value={coupon} onChange={e => setCoupon(e.target.value)} placeholder='Coupon code' />
<button onClick={applyCoupon}>Apply</button>
{discount > 0 && <p>Discount: {discount}% off!</p>}
<p><strong>Total: ${total.toFixed(2)}</strong></p>
<button onClick={() => setCheckout(true)}>Checkout</button>
</div>
);
}
```

```
$ Output
Initial state: Empty cart, subtotal $0
Add Laptop: Cart = [{ id:1, name:"Laptop", price:999, qty:1 }], total $999
Add Mouse: Cart = [Laptop x1, Mouse x1], total $1028
Add Laptop: Cart = [Laptop x2, Mouse x1], total $2027
Apply "SAVE20": Discount 20%, total $1621.60
Click Checkout: Shows "■ Order placed! Total: $1621.60"
```

5.5 useEffect — Side Effects

■ *Real-world: Search with debouncing and API call*

```
import { useState, useEffect, useRef } from 'react';

function ProductSearch() {
const [query, setQuery] = useState('');
const [results, setResults] = useState([]);
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);
const debounceRef = useRef(null);

// Effect 1: Debounced search
useEffect(() => {
// Clear previous timer
if (debounceRef.current) clearTimeout(debounceRef.current);

if (!query.trim()) {
setResults([]);
return;
}

// Wait 500ms after user stops typing
debounceRef.current = setTimeout(async () => {
setLoading(true);
setError(null);
try {
const res = await fetch(`/api/products?search=${query}`);
if (!res.ok) throw new Error('Search failed');
const data = await res.json();
setResults(data.data);
} catch (err) {
setError(err.message);
} finally {
setLoading(false);
}
}, 500);

// Cleanup function: cancel timer when component unmounts or query changes
return () => clearTimeout(debounceRef.current);
}, [query]); // Re-run when 'query' changes

// Effect 2: Update document title
useEffect(() => {
document.title = query ? `Search: ${query}` : 'My Shop';
return () => { document.title = 'My Shop'; }; // Cleanup
}, [query]);

// Effect 3: Run once on mount
useEffect(() => {
console.log('Component mounted!');
return () => console.log('Component unmounted!'); // Cleanup
}, []); // Empty array = run once

return (
<div>
<input
value={query}
onChange={e => setQuery(e.target.value)}
placeholder='Search products...'
/>
{loading && <p>Searching...</p>}
```

```
{error && <p style={{color:'red'}}>{error}</p>}
<ul>
{results.map(p => <li key={p.id}>{p.name} — ${p.price}</li>)}
</ul>
</div>
);
}
```

```
$ Output
Component mounted!
User types "lap": waits 500ms, then → GET /api/products?search=lap
User types "lapto": cancels previous, waits 500ms, → GET /api/products?search=lapto
Results rendered: • Laptop — $999
User clears input: Results cleared, title resets to "My Shop"
```

5.6 useReducer — Complex State Management

■ *Real-world: Shopping cart with useReducer (better for complex state)*

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

// Define all possible state changes
function cartReducer(state, action) {
switch (action.type) {

case 'ADD_ITEM': {
const exists = state.items.find(i => i.id === action.item.id);
const items = exists
? state.items.map(i => i.id === action.item.id ? { ...i, qty: i.qty + 1 } : i)
: [...state.items, { ...action.item, qty: 1 }];
return { ...state, items };
}

case 'REMOVE_ITEM':
return { ...state, items: state.items.filter(i => i.id !== action.id) };

case 'UPDATE_QTY':
return { ...state, items: state.items.map(i =>
i.id === action.id ? { ...i, qty: Math.max(1, action.qty) } : i
)};

case 'CLEAR_CART':
return { ...state, items: [] };

case 'APPLY_COUPON':
return { ...state, discount: action.discount };

default:
return state;
}
}

const initialState = { items: [], discount: 0 };

function Cart() {
const [state, dispatch] = useReducer(cartReducer, initialState);

// Dispatch actions
const add = item => dispatch({ type: 'ADD_ITEM', item });
const remove = id => dispatch({ type: 'REMOVE_ITEM', id });
const clear = () => dispatch({ type: 'CLEAR_CART' });
const coupon = () => dispatch({ type: 'APPLY_COUPON', discount: 15 });

const total = state.items.reduce((sum, i) => sum + i.price * i.qty, 0);

return (
<div>
<button onClick={() => add({ id: 1, name: 'Laptop', price: 999 })}>
Add Laptop
</button>
<button onClick={coupon}>Apply 15% Coupon</button>
<button onClick={clear}>Clear Cart</button>
<p>Items: {state.items.length} | Discount: {state.discount}%</p>
<p>Total: ${(total * (1 - state.discount/100)).toFixed(2)}</p>
</div>
);
}
```

5.7 useContext — Global State

■ *Real-world: Auth context for the entire app*

```
import { createContext, useContext, useState, useEffect } from 'react';

// 1. Create the context
const AuthContext = createContext(null);

// 2. Create a provider component
export function AuthProvider({ children }) {
const [user, setUser] = useState(null);
const [loading, setLoading] = useState(true);

// Check if user is already logged in (from localStorage)
useEffect(() => {
const token = localStorage.getItem('token');
if (token) {
fetchUserProfile(token).then(setUser).finally(() => setLoading(false));
} else {
setLoading(false);
}
}, []);

const login = async (email, password) => {
const res = await fetch('/api/auth/login', {
method: 'POST',
headers: { 'Content-Type': 'application/json' },
body: JSON.stringify({ email, password })
});
const data = await res.json();
if (!res.ok) throw new Error(data.error);

localStorage.setItem('token', data.token);
setUser(data.user);
return data.user;
};

const logout = () => {
localStorage.removeItem('token');
setUser(null);
};

return (
<AuthContext.Provider value={{ user, login, logout, loading }}>
{children}
</AuthContext.Provider>
);
}

// 3. Custom hook for easy access
export function useAuth() {
const ctx = useContext(AuthContext);
if (!ctx) throw new Error('useAuth must be used within AuthProvider');
return ctx;
}

// 4. Wrap your app
// index.js
ReactDOM.render(
<AuthProvider>
<App />
</AuthProvider>,
```

```
document.getElementById('root')
);

// 5. Use in any component
function Navbar() {
  const { user, logout } = useAuth();
  return (
    <nav>
      {user ? (
        <>
          <span>Hello, {user.name}!</span>
          <button onClick={logout}>Logout</button>
        </>
      ) : (
        <a href='/login'>Login</a>
      )}
    </nav>
  );
}
```

```
$ Output
# When user IS logged in:
<nav>  Hello, Alice!  [Logout button]  </nav>

# When user is NOT logged in:
<nav>  <Login link>  </nav>

# useAuth() throws if used outside <AuthProvider>:
Error: useAuth must be used within AuthProvider
```

useContext Pattern Summary

1. createContext(null) — create the context object
2. <Context.Provider value={…}> — wrap your app/subtree
3. useContext(Context) — consume in any child component
4. Custom hook (useAuth) — hide the useContext call for cleaner code

Any component inside <AuthProvider> can call useAuth() to get
{ user, login, logout, loading } without prop drilling.

5.8 Custom Hooks — Reusable Logic

■ *Real-world: useLocalStorage, useFetch, useForm hooks*

```
// ■■ useLocalStorage ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
function useLocalStorage(key, initialValue) {
const [value, setValue] = useState(() => {
try {
const stored = localStorage.getItem(key);
return stored ? JSON.parse(stored) : initialValue;
} catch {
return initialValue;
}
});

const setStoredValue = (newValue) => {
setValue(newValue);
localStorage.setItem(key, JSON.stringify(newValue));
};

return [value, setStoredValue];
}

// Usage:
const [theme, setTheme] = useLocalStorage('theme', 'light');
// Persists across page refreshes!

// ■■ useFetch ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
function useFetch(url, options = {}) {
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);

useEffect(() => {
const controller = new AbortController(); // Cancel fetch on unmount

async function fetchData() {
try {
setLoading(true);
const res = await fetch(url, { ...options, signal: controller.signal });
if (!res.ok) throw new Error(`HTTP ${res.status}`);
setData(await res.json());
} catch (err) {
if (err.name !== 'AbortError') setError(err.message);
} finally {
setLoading(false);
}
}

fetchData();
return () => controller.abort(); // Cleanup
}, [url]);

return { data, loading, error };
}

// Usage:
function UserProfile({ userId }) {
const { data, loading, error } = useFetch(`/api/users/${userId}`);
if (loading) return <Spinner />;
if (error) return <p>Error: {error}</p>;
return <div>{data?.user.name}</div>;
}
```

5.8 Custom Hooks — Expected Outputs

■ *useLocalStorage hook — behaviour across page refreshes*

```
$ Output
const [theme, setTheme] = useLocalStorage('theme', 'light');

# Initial render (key not in localStorage yet):
theme  →  'light'     (uses initialValue)

# After  setTheme('dark'):
theme  →  'dark'
localStorage.getItem('theme')  →  '"dark"'

# After page refresh:
theme  →  'dark'     (read back from localStorage ✓)
```

■ *useFetch hook — loading, data, and error states*

```
$ Output
// Immediately after mount:
{ data: null, loading: true,  error: null }

// After successful fetch  GET /api/users/1:
{ data: { user: { id:1, name:'Alice' } }, loading: false, error: null }

// After a failed fetch  (network error or 4xx/5xx):
{ data: null, loading: false, error: 'HTTP 404' }

// When userId prop changes → re-fetches automatically
// Previous fetch is aborted via AbortController (no stale data)
```

Why Custom Hooks?

Custom hooks let you extract stateful logic into reusable functions.

useLocalStorage → any component can persist state across refreshes.

useFetch → any component gets loading/error/data handling for free.

The Rules of Hooks still apply: only call hooks at the top level,

only call them from React functions or other hooks.

Chapter 6: Amazon Web Services (AWS)

AWS is like renting an entire city of technology infrastructure instead of building it yourself. Instead of buying servers, storage, and networking equipment, you rent exactly what you need and pay only for what you use.

■ **Cloud vs Traditional Hosting**
Traditional: Buy a server for $5,000, host in your office, manage it yourself.
AWS: Rent computing power for $0.023/hour. Scale up in seconds. Zero hardware.
AWS handles: power, cooling, hardware failures, security patches, network.
You handle: your application code and data.

6.1 AWS Global Infrastructure

- Regions: Geographic areas with multiple data centers (us-east-1, eu-west-1, ap-south-1, etc.)
- Availability Zones (AZs): Isolated data centers within a region. Deploy across multiple AZs for high availability.
- Edge Locations: Smaller locations for CDN caching (CloudFront). 400+ worldwide.
- Best practice: Deploy to at least 2 AZs within a region for production apps.

6.2 IAM — Identity and Access Management

IAM controls who can do what in your AWS account. This is the most important security service.

■ *Real-world: Creating a deployment user with minimal permissions*

```
// IAM Policy — allows only specific S3 actions on specific bucket
{
"Version": "2012-10-17",
"Statement": [
{
"Sid": "S3DeployAccess",
"Effect": "Allow",
"Action": [
"s3:PutObject",
"s3:GetObject",
"s3:DeleteObject",
"s3:ListBucket"
],
"Resource": [
"arn:aws:s3:::my-react-app",
"arn:aws:s3:::my-react-app/*"
]
},
{
"Sid": "CloudFrontInvalidate",
"Effect": "Allow",
"Action": "cloudfront:CreateInvalidation",
"Resource": "arn:aws:cloudfront::123456789:distribution/E1EXAMPLE"
}
]
}

// Golden Rules of IAM:
// 1. Principle of Least Privilege — only grant what's needed
// 2. Never use root account credentials in code
// 3. Use IAM Roles for EC2/Lambda (not access keys!)
// 4. Enable MFA on all accounts
// 5. Rotate access keys regularly
```

6.3 EC2 — Elastic Compute Cloud

EC2 gives you virtual machines in the cloud. You choose the OS, hardware specs (CPU, RAM), and pay by the hour.

■ *Common instance types*

Family	Use Case	Example	vCPU/RAM
t3/t4g	General purpose (cheap, burstable)	Web servers, dev environments	t3.micro: 2 vCPU, 1GB RAM
m6i/m7i	Balanced CPU/memory	Application servers	m6i.large: 2 vCPU, 8GB RAM
c6i/c7i	Compute optimized	CPU-intensive processing	c6i.xlarge: 4 vCPU, 8GB RAM
r6i/r7i	Memory optimized	In-memory databases, caching	r6i.large: 2 vCPU, 16GB RAM
g4dn/p3	GPU instances	ML training, video encoding	g4dn.xlarge: 4 vCPU + NVIDIA T4

■ *Launching and connecting to EC2 via AWS CLI*

```
# Install AWS CLI
# https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html

# Configure credentials
aws configure
# AWS Access Key ID: AKIAIOSFODNN7EXAMPLE
# AWS Secret Access Key: wJalrXUtnFEMI/K7MDENG...
# Default region: us-east-1
# Output format: json

# Launch EC2 instance
aws ec2 run-instances \
--image-id ami-0c02fb55956c7d316 \ # Amazon Linux 2023
--instance-type t3.micro \
--key-name MyKeyPair \ # SSH key for login
--security-groups my-sg \
--user-data file://startup.sh # Script to run on first boot

# startup.sh — auto-configure the server on launch
#!/bin/bash
yum update -y
yum install -y nodejs npm
npm install -g pm2
cd /home/ec2-user
git clone https://github.com/myrepo/api.git
cd api && npm install
pm2 start server.js --name 'api'
pm2 startup && pm2 save # Start on system reboot

# SSH into the instance
ssh -i MyKeyPair.pem ec2-user@52.23.45.67
```

6.4 S3 — Simple Storage Service

S3 stores any type of file (images, videos, backups, code) with 99.999999999% (11 nines) durability. It's also used to host static websites and React apps.

■ *Complete S3 operations with Node.js AWS SDK*

```
npm install @aws-sdk/client-s3 @aws-sdk/s3-request-presigner

const {
S3Client,
PutObjectCommand,
GetObjectCommand,
DeleteObjectCommand,
ListObjectsV2Command,
CopyObjectCommand
} = require('@aws-sdk/client-s3');
const { getSignedUrl } = require('@aws-sdk/s3-request-presigner');

const s3 = new S3Client({ region: 'us-east-1' });
const BUCKET = 'my-app-storage';

// ■■ UPLOAD ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
async function uploadFile(key, fileBuffer, contentType) {
await s3.send(new PutObjectCommand({
Bucket: BUCKET,
Key: key, // e.g. 'avatars/user-42.jpg'
Body: fileBuffer,
ContentType: contentType, // 'image/jpeg'
Metadata: { uploadedBy: 'api-server' }
}));
return `https://${BUCKET}.s3.amazonaws.com/${key}`;
}

// ■■ GENERATE PRE-SIGNED URL (allow direct upload from browser) ■■
async function getUploadUrl(key, contentType) {
const command = new PutObjectCommand({
Bucket: BUCKET, Key: key, ContentType: contentType
});
// URL valid for 15 minutes
const url = await getSignedUrl(s3, command, { expiresIn: 900 });
return url;
}

// ■■ DOWNLOAD ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
async function downloadFile(key) {
const { Body, ContentType } = await s3.send(new GetObjectCommand({
Bucket: BUCKET, Key: key
}));
const content = await Body.transformToString();
return { content, contentType: ContentType };
}

// ■■ LIST FILES ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
async function listFiles(prefix = '') {
const { Contents } = await s3.send(new ListObjectsV2Command({
Bucket: BUCKET, Prefix: prefix
}));
return Contents?.map(obj => ({
key: obj.Key,
size: `${(obj.Size / 1024).toFixed(1)} KB`,
lastModified: obj.LastModified
})) || [];
```

```
}

// ■■ DELETE ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
async function deleteFile(key) {
await s3.send(new DeleteObjectCommand({ Bucket: BUCKET, Key: key }));
}

// ■■ Express endpoint for avatar upload ■■■■■■■■■■■■■■■■■■■■■■
app.post('/api/avatar/upload-url', authenticate, async (req, res) => {
const { filename, contentType } = req.body;
const key = `avatars/${req.user.userId}/${Date.now()}-${filename}`;
const url = await getUploadUrl(key, contentType);
res.json({ uploadUrl: url, key });
});
```

```
$ Output
POST /api/avatar/upload-url
→ {
"uploadUrl": "https://my-app-storage.s3.amazonaws.com/avatars/42/...",
"key": "avatars/42/1705320000-profile.jpg"
}

# Browser then uploads directly to S3 with PUT request to uploadUrl
# This avoids routing large files through your server!
```

6.5 Lambda — Serverless Functions

Lambda runs your code without you managing servers. You upload a function, define what triggers it (HTTP request, S3 upload, schedule), and AWS runs it automatically. You pay only when it runs (per millisecond).

■ *Real-world: Image resizing Lambda triggered by S3 upload*

```
// lambda/resize-image.js
const { S3Client, GetObjectCommand, PutObjectCommand } = require('@aws-sdk/client-s3');
const sharp = require('sharp'); // Image processing library

const s3 = new S3Client({ region: 'us-east-1' });

exports.handler = async (event) => {
// S3 event tells us which file was uploaded
const bucket = event.Records[0].s3.bucket.name;
const key = decodeURIComponent(event.Records[0].s3.object.key);

console.log(`Processing: s3://${bucket}/${key}`);

// Only process images in 'uploads/' folder
if (!key.startsWith('uploads/')) return;

try {
// Download original image
const { Body } = await s3.send(new GetObjectCommand({ Bucket: bucket, Key: key }));
const original = await Body.transformToByteArray();

// Resize to 3 sizes
const sizes = [
{ name: 'thumb', width: 150, height: 150 },
{ name: 'medium', width: 800, height: 600 },
{ name: 'large', width: 1920, height: 1080 },
];

await Promise.all(sizes.map(async ({ name, width, height }) => {
const resized = await sharp(original)
.resize(width, height, { fit: 'inside', withoutEnlargement: true })
.jpeg({ quality: 80 })
.toBuffer();

const newKey = key.replace('uploads/', `${name}/`).replace(/\.[^.]+$/, '.jpg');

await s3.send(new PutObjectCommand({
Bucket: bucket,
Key: newKey,
Body: resized,
ContentType: 'image/jpeg'
}));
console.log(`Created ${name}: ${newKey}`);
}));

return { statusCode: 200, body: 'Success' };

} catch (error) {
console.error('Error:', error);
throw error; // Lambda will retry
}
};
```

```
$ Output
# Trigger: User uploads uploads/photo.jpg to S3
Processing: s3://my-bucket/uploads/photo.jpg
```

```
Created thumb: thumb/photo.jpg (150x150)
Created medium: medium/photo.jpg (800x600)
Created large: large/photo.jpg (1920x1080)

# CloudWatch Logs:
START RequestId: abc123 Version: $LATEST
END RequestId: abc123
REPORT Duration: 890ms | Memory: 512MB | Cost: $0.000015
```

6.6 RDS — Relational Database Service

RDS manages relational databases (MySQL, PostgreSQL, MariaDB, SQL Server, Oracle) for you. It handles backups, patching, replication, and failover automatically.

■ *Connect to RDS PostgreSQL with connection pooling*

```
npm install pg

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

// Connection pool — reuse connections efficiently
const pool = new Pool({
host: process.env.RDS_HOST,
port: 5432,
database: 'shopdb',
user: process.env.RDS_USER,
password: process.env.RDS_PASSWORD,
max: 20, // Max pool size
idleTimeoutMillis: 30000,
connectionTimeoutMillis: 2000,
ssl: { rejectUnauthorized: false }
});

// Wrapper with automatic connection management
async function query(sql, params = []) {
const client = await pool.connect();
try {
const result = await client.query(sql, params);
return result.rows;
} finally {
client.release(); // Always release back to pool
}
}

// Example queries
async function getUserOrders(userId) {
// Parameterized query — prevents SQL injection!
const orders = await query(`
SELECT o.id, o.total, o.status, o.created_at,
COUNT(oi.id) as item_count
FROM orders o
JOIN order_items oi ON oi.order_id = o.id
WHERE o.user_id = $1
GROUP BY o.id
ORDER BY o.created_at DESC
LIMIT 20
`, [userId]);
return orders;
}

async function createOrder(userId, items) {
const client = await pool.connect();
try {
await client.query('BEGIN');

const [order] = await client.query(
'INSERT INTO orders (user_id, status) VALUES ($1, $2) RETURNING id'
[userId, 'pending']
);

for (const item of items) {
await client.query(
```

```
  await client.query('INSERT INTO order_items ...',
    [order.id, item.productId, item.qty, item.price]
  );
}
await client.query('COMMIT');
return order.id;
} catch (err) {
  await client.query('ROLLBACK'); // Undo on error
  throw err;
} finally {
  client.release();
}
}
```

6.7 CloudFront — Content Delivery Network

CloudFront is AWS's CDN. It caches your content at 400+ edge locations worldwide so users get data from a server near them instead of your origin (potentially seconds away).

■ *Setting up CloudFront for React app + API*

```
# CloudFront Distribution (simplified):

Origins:
  1. S3 Bucket   → serves React app (HTML/JS/CSS)
  2. API Gateway → serves REST API

Behaviors (path-based routing):
  /api/* → forward to API Gateway (no caching)
  /*     → serve from S3 (cache 1 year for hashed assets)

# Cache-Control headers for React build:
# Hashed files (main.abc123.js): cache forever
# index.html: never cache (new deployments work instantly)

# Invalidate after deployment:
aws cloudfront create-invalidation \
  --distribution-id E1EXAMPLE \
  --paths '/index.html' '/*.html'

# Benefits:
# User in India served from Mumbai edge = 10ms
# Without CDN, served from us-east-1  = 250ms
# Reduces S3 costs (CloudFront caches, fewer S3 req)
# DDoS protection (CloudFront absorbs traffic)
```

```
$ Output
$ aws cloudfront create-invalidation \
    --distribution-id E1EXAMPLE --paths '/index.html'

{
    "Invalidation": {
        "Id": "I2J3K4L5M6N7O8",
        "Status": "InProgress"
    }
}

# After invalidation completes (~30 seconds):
# All 400+ edge locations serve the fresh index.html
# Cached JS/CSS assets (hashed filenames) never need invalidation
```

6.8 SQS & SNS — Messaging Services

■ *Real-world: Order processing with SQS queue*

```
const { SQSClient, SendMessageCommand,
ReceiveMessageCommand, DeleteMessageCommand } = require('@aws-sdk/client-sqs');

const sqs = new SQSClient({ region: 'us-east-1' });
const QUEUE = 'https://sqs.us-east-1.amazonaws.com/123456789/orders-queue';

// ■■ PRODUCER: API creates orders and sends to queue ■■■■■■■■
app.post('/api/orders', authenticate, async (req, res) => {
const order = await Order.create({ ...req.body, userId: req.user.userId });

// Send to SQS for async processing
await sqs.send(new SendMessageCommand({
QueueUrl: QUEUE,
MessageBody: JSON.stringify({ orderId: order.id, userId: req.user.userId }),
MessageDeduplicationId: `order-${order.id}`, // For FIFO queues
MessageGroupId: 'orders',
DelaySeconds: 0
}));

res.status(202).json({ // 202 = Accepted (processing async)
success: true,
orderId: order.id,
message: 'Order received and being processed'
});
});

// ■■ CONSUMER: Background worker processes the queue ■■■■■■■■
async function processOrders() {
console.log('Order worker started. Waiting for messages...');

while (true) { // Run forever
const { Messages } = await sqs.send(new ReceiveMessageCommand({
QueueUrl: QUEUE,
MaxNumberOfMessages: 5,
WaitTimeSeconds: 20, // Long polling — wait up to 20s
}));

if (!Messages || Messages.length === 0) continue;

await Promise.all(Messages.map(async (msg) => {
const { orderId, userId } = JSON.parse(msg.Body);

try {
console.log(`Processing order #${orderId}...`);
await chargePayment(orderId);
await reserveInventory(orderId);
await sendConfirmationEmail(userId, orderId);
await Order.updateStatus(orderId, 'confirmed');

// Delete from queue (mark as done)
await sqs.send(new DeleteMessageCommand({
QueueUrl: QUEUE,
ReceiptHandle: msg.ReceiptHandle
}));
console.log(`Order #${orderId} completed`);

} catch (err) {
console.error(`Order #${orderId} failed:`, err.message);
// Don't delete — message returns to queue after visibility timeout
```

```
        // SQS will retry automatically!
      }
    }));
  }
}

processOrders(); // Run in separate process/Lambda
```

```
$ Output
POST /api/orders → 202 Accepted
{ "orderId": 1001, "message": "Order received and being processed" }

# Worker process output:
Order worker started. Waiting for messages...
Processing order #1001...
→ Payment charged
→ Inventory reserved
→ Email sent to alice@example.com
Order #1001 completed
```

6.9 Secrets Manager

■ *Store and retrieve sensitive credentials securely*

```
npm install @aws-sdk/client-secrets-manager
```

```
const { SecretsManagerClient,
  GetSecretValueCommand } = require('@aws-sdk/client-secrets-manager');

const secretsClient = new SecretsManagerClient({ region: 'us-east-1' });

async function getSecret(secretName) {
  const { SecretString } = await secretsClient.send(
    new GetSecretValueCommand({ SecretId: secretName })
  );
  return JSON.parse(SecretString);
}

// Usage — fetch all secrets at startup
async function initializeApp() {
  const dbCredentials = await getSecret('prod/myapp/db');
  const apiKeys       = await getSecret('prod/myapp/apis');

  const dbPool = new Pool({
    host:     dbCredentials.host,
    user:     dbCredentials.username,
    password: dbCredentials.password,
    database: dbCredentials.dbname
  });

  console.log('App initialized securely');
  return { dbPool, stripeKey: apiKeys.stripeSecretKey };
}
```

```
$ Output
# On startup — secrets fetched from AWS:
App initialized securely

# dbPool is ready with credentials from Secrets Manager
# stripeKey  →  'sk_live_...'  (never hard-coded in source)

# If secret does not exist or IAM permission is missing:
ResourceNotFoundException: Secrets Manager can't find the specified secret.

# Tip: Rotate secrets in the console — no code change needed.
```

6.10 CloudWatch — Monitoring and Logging

■ *Structured logging for CloudWatch Logs Insights*

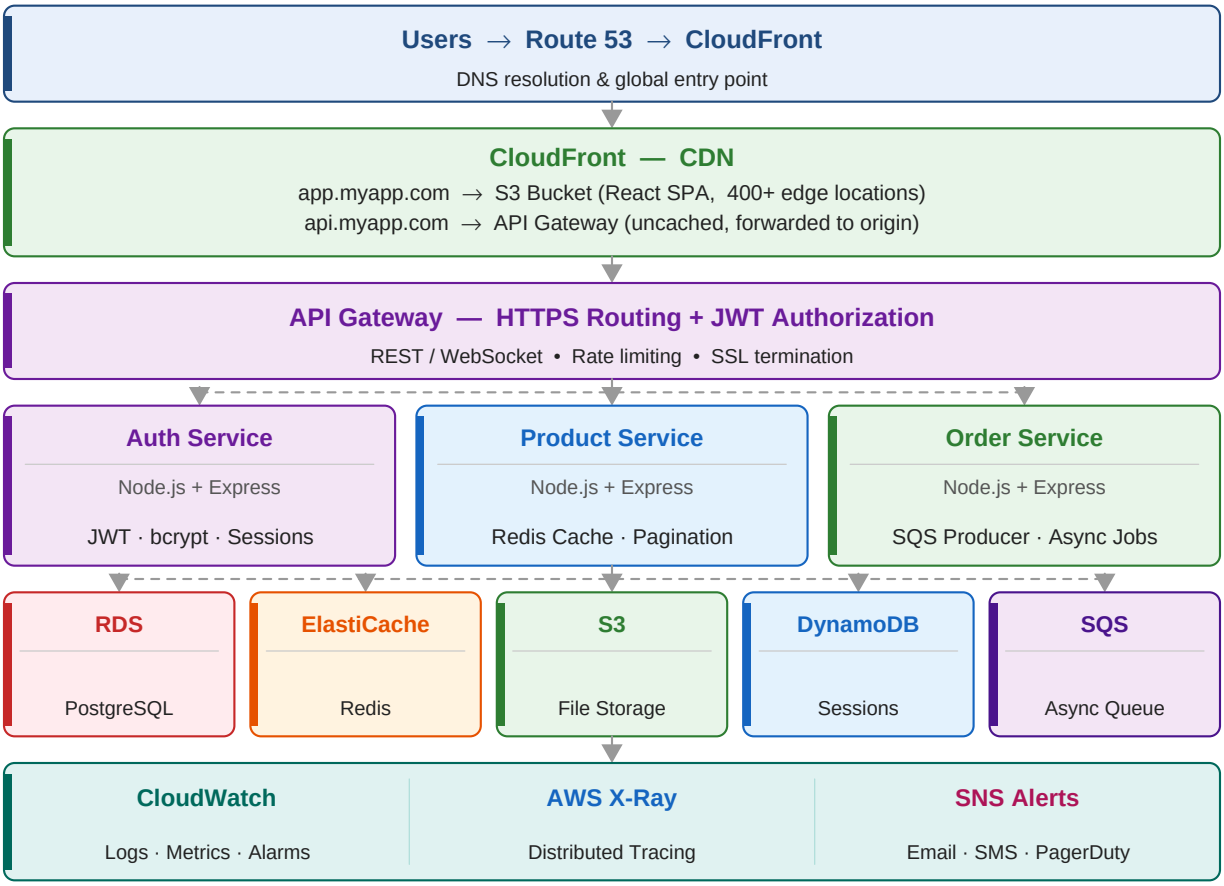
```
// Structured logging (JSON format — searchable in CloudWatch)
const logger = {
info: (msg, data = {}) => console.log(JSON.stringify({ level: 'INFO', msg, ...data, ts: new
Date().toISOString() })),
error: (msg, data = {}) => console.log(JSON.stringify({ level: 'ERROR', msg, ...data, ts: new
Date().toISOString() })),
warn: (msg, data = {}) => console.log(JSON.stringify({ level: 'WARN', msg, ...data, ts: new
Date().toISOString() })),
};

// Usage in your API
app.post('/api/orders', authenticate, async (req, res) => {
const start = Date.now();
try {
const order = await createOrder(req.body);
logger.info('Order created', { orderId: order.id, userId: req.user.userId, duration: Date.now() -
start });
res.status(201).json({ data: order });
} catch (err) {
logger.error('Order creation failed', { error: err.message, userId: req.user.userId });
res.status(500).json({ error: 'Order failed' });
}
});

// CloudWatch Logs Insights query:
// fields @timestamp, msg, orderId, duration
// | filter level = "ERROR"
// | sort @timestamp desc
// | limit 20
```

6.11 Complete Full-Stack Architecture

■ *Production-grade architecture for a SaaS application*



Chapter 7: Putting It All Together

This chapter shows how Node.js, Express, Redis, React, and AWS work together in a real e-commerce application — from a caching-aware Express backend to a React API service layer.

7.1 Backend: Express + Redis + RDS

■ *Complete API with caching, auth, and DB*

```
// server.js — Complete production-ready setup
require('dotenv').config();
const express = require('express');
const redis   = require('redis');
const { Pool } = require('pg');
const jwt     = require('jsonwebtoken');
const app  = express();
const PORT = process.env.PORT || 3000;

// ■■ Database ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
const db = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: { rejectUnauthorized: false }
});

// ■■ Redis ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
const cache = redis.createClient({ url: process.env.REDIS_URL });
cache.connect().then(() => console.log('Redis connected'));

// ■■ Middleware ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
app.use(express.json());

// Auth middleware — JWT verification + Redis blacklist check
const auth = async (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(401).json({ error: 'No token' });
  try {
    const blacklisted = await cache.get(`blacklist:${token}`);
    if (blacklisted) return res.status(401).json({ error: 'Token revoked' });
    req.user = jwt.verify(token, process.env.JWT_SECRET);
    next();
  } catch {
    res.status(403).json({ error: 'Invalid token' });
  }
};

// ■■ Routes ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
app.get('/api/products', async (req, res) => {
  const cached = await cache.get('products:all');
  if (cached) return res.json(JSON.parse(cached));
  const { rows } = await db.query('SELECT * FROM products ORDER BY name');
  await cache.setEx('products:all', 600, JSON.stringify(rows));
  res.json(rows);
});

app.post('/api/auth/logout', auth, async (req, res) => {
  const token = req.headers.authorization.split(' ')[1];
  const exp   = req.user.exp - Math.floor(Date.now() / 1000);
  await cache.setEx(`blacklist:${token}`, exp, '1');
  res.json({ message: 'Logged out' });
});

app.listen(PORT, () => console.log(`API running on port ${PORT}`));
```

7.2 Frontend: React calling the API

■ *Complete API service layer for React*

```
// src/services/api.js
const BASE_URL = process.env.REACT_APP_API_URL;

class ApiService {
  constructor() {
    this.token = localStorage.getItem('token');
  }

  setToken(token) {
    this.token = token;
    localStorage.setItem('token', token);
  }

  clearToken() {
    this.token = null;
    localStorage.removeItem('token');
  }

  async request(method, endpoint, body = null) {
    const headers = { 'Content-Type': 'application/json' };
    if (this.token) headers['Authorization'] = `Bearer ${this.token}`;
    const res = await fetch(`${BASE_URL}${endpoint}`, {
      method, headers,
      body: body ? JSON.stringify(body) : null
    });
    if (res.status === 401) {
      this.clearToken();
      window.location.href = '/login';  // Redirect on expired token
      return;
    }
    const data = await res.json();
    if (!res.ok) throw new Error(data.error || 'Something went wrong');
    return data;
  }

  // Auth
  login    = (email, pw)        => this.request('POST', '/api/auth/login',    { email, password: pw });
  register = (name, email, pw)  => this.request('POST', '/api/auth/register', { name, email, password: pw });
  logout   = ()                 => this.request('POST', '/api/auth/logout');

  // Products
  getProducts = ()    => this.request('GET', '/api/products');
  getProduct  = (id)  => this.request('GET', `/api/products/${id}`);

  // Orders
  createOrder = (data) => this.request('POST', '/api/orders', data);
  getOrders   = ()     => this.request('GET',  '/api/orders');
}

export const api = new ApiService();

// Usage in components:
const products        = await api.getProducts();
const { token, user } = await api.login('alice@example.com', 'password123');
api.setToken(token);
```

7.3 Environment Configuration

■ *.env file — keep all config in one place*

```
# Backend .env
NODE_ENV=production
PORT=3000

# Database
DATABASE_URL=postgresql://user:pass@mydb.xyz.us-east-1.rds.amazonaws.com:5432/shopdb

# Redis
REDIS_URL=redis://myredis.xyz.cache.amazonaws.com:6379

# Auth
JWT_SECRET=your-long-random-secret-at-least-64-chars-long!!

# AWS
AWS_REGION=us-east-1
S3_BUCKET=my-shop-uploads

# Frontend .env
REACT_APP_API_URL=https://api.myshop.com
REACT_APP_STRIPE_KEY=pk_live_xxxxx
```

7.4 Deployment Checklist

- Build React app: npm run build
- Run tests: npm test (all must pass)
- Upload frontend to S3: aws s3 sync build/ s3://my-react-app --delete
- Invalidate CloudFront cache: aws cloudfront create-invalidation ...
- Deploy backend Docker image to ECR (Elastic Container Registry)
- Update ECS service to use new image (zero-downtime rolling update)
- Run database migrations: npm run migrate
- Verify health check endpoint: GET /api/health → 200 OK
- Monitor CloudWatch logs for errors in the first 30 minutes
- Set up CloudWatch alarms: Error rate > 1%, P99 latency > 500ms

Quick Reference Card

Node.js Cheatsheet

```
// Core methods
require('module') // Import a module
module.exports = value // Export from a module
process.env.KEY // Access environment variable
process.exit(0) // Exit process (0 = success)
__dirname // Directory of current file
__filename // Path of current file

// Async patterns
const result = await asyncFn() // Wait for promise
Promise.all([p1, p2]) // Run in parallel, wait for all
Promise.race([p1, p2]) // Return first one to finish
Promise.allSettled([p1, p2]) // Wait for all, even failures
```

Express Cheatsheet

```
app.use(middleware) // Apply to all routes
app.use('/path', middleware) // Apply to specific path
app.get/post/put/patch/delete // Route handlers
req.params.id // URL parameter (/users/:id)
req.query.page // Query string (?page=2)
req.body.email // Request body (needs express.json())
req.headers['authorization'] // Request header
res.json(data) // Send JSON response
res.status(404).json({error}) // Send with status code
res.status(204).send() // Empty response
next() // Pass to next middleware
next(error) // Pass to error handler
```

Redis Cheatsheet

```
// Strings
client.set(key, value) // Set key
client.get(key) // Get key
client.setEx(key, ttl, value) // Set with expiry
client.del(key) // Delete key
client.incr(key) // Increment counter

// Hashes
client.hSet(key, field, val) // Set hash field
client.hGet(key, field) // Get hash field
client.hGetAll(key) // Get all fields

// Lists
client.rPush(key, value) // Add to end
client.lPop(key) // Remove from front
client.lRange(key, 0, -1) // Get all items

// Sets
client.sAdd(key, member) // Add member
client.sMembers(key) // Get all members
client.sIsMember(key, member) // Check membership

// Sorted Sets
client.zAdd(key, [{score,value}]) // Add with score
client.zRangeWithScores(key,0,-1) // Get ranked list
```

React Hooks Cheatsheet

```
useState(initial) // Local component state
useEffect(fn, [deps]) // Side effects (fetch, subscriptions)
useContext(Context) // Access global context
useReducer(reducer, state) // Complex state management
useRef(initial) // Mutable ref, doesn't trigger render
useMemo(() => val, [deps]) // Memoize expensive computation
useCallback(fn, [deps]) // Memoize function reference

// useEffect patterns
useEffect(() => {}, []) // Run once (mount)
useEffect(() => {}, [value]) // Run when value changes
useEffect(() => () => cleanup()) // With cleanup function
```

AWS CLI Cheatsheet

```
# Configure
aws configure # Set credentials
aws sts get-caller-identity # Who am I?

# S3
aws s3 ls s3://my-bucket # List files
aws s3 cp file.txt s3://my-bucket/ # Upload
aws s3 sync ./build s3://my-bucket --delete # Sync folder
aws s3 rm s3://my-bucket/file.txt # Delete

# EC2
aws ec2 describe-instances # List instances
aws ec2 start-instances --instance-ids i-123 # Start
aws ec2 stop-instances --instance-ids i-123 # Stop

# Lambda
aws lambda list-functions # List functions
aws lambda invoke --function-name MyFn out.json # Invoke

# CloudWatch
aws logs tail /aws/lambda/MyFunction --follow # Stream logs
```

■ You've reached the end! Keep building, keep learning.

Node.js · REST · Express · Redis · React · AWS