Backend Development Roadmap

9-Week Intensive Training Program

Node.js + Express + PostgreSQL + Redis

Generated: October 04, 2025

Week 1: HTTP Fundamentals, Simple CRUD API, Tests

■ Focus

Goal: Master how HTTP works + build + test a basic CRUD API using Express.

Core Concepts (80/20):

- • HTTP methods: GET, POST, PUT, DELETE
- • Status codes, headers, request/response flow
- • Express routing & middleware
- • JSON parsing
- • RESTful CRUD API design
- • Basic error handling
- • Writing & testing endpoints (with Jest / Supertest)

■ SIMPLE LEVEL - HTTP + Basic Express Setup

Goal: Understand HTTP fundamentals + handle routes & responses.

1. Create a simple Express server that responds 'Hello World' on / route

Concept: Basic HTTP GET

2. Add routes /about, /contact, each returning JSON {message: '...'}

Concept: Routing & JSON

3. Log each incoming request's method, URL, and time to console

Concept: Middleware basics

4. Serve static HTML/CSS files using Express.static()

Concept: Static serving

5. Experiment with HTTP status codes (res.status(404).send('Not Found'))

Concept: HTTP fundamentals

6. Send custom headers (res.set('X-Powered-By', 'NodeJS'))

Concept: Headers

■ Outcome: You can build and reason about simple HTTP responses and understand the client-server lifecycle.

■ INTERMEDIATE LEVEL - CRUD API

Goal: Implement a RESTful API with in-memory storage.

1. Create an Express app /api/users with CRUD routes

Concept: REST patterns

2. Implement in-memory storage (array of users with id, name, email)

Concept: Data handling

3. Implement each: GET /users → list all, GET /users/:id → get one, POST /users → create new, PUT /users/:id → update, DELETE /users/:id → delete

Concept: Full CRUD

4. Add validation (check if email exists before adding)

Concept: Input validation

5. Return proper HTTP codes: 201 for creation, 404 for not found, etc.

Concept: REST conventions

6. Add middleware for error handling (next(err))

Concept: Error flow

7. Create a .env file for config (PORT, etc.) using dotenv

Concept: Environment setup

■ Outcome: You can design and code a REST API with proper structure, validation, and HTTP conventions.

■ HARD LEVEL - Testing + Robustness

Goal: Make your API production-grade in miniature.

1. Write integration tests for CRUD routes using Jest + Supertest

Concept: Automated testing

2. Refactor routes into separate files (routes/, controllers/)

Concept: Code organization

3. Add centralized error handling middleware

Concept: Clean error structure

4. Simulate database delays using setTimeout and handle async with try/catch

Concept: Async control

5. Add a custom middleware to rate-limit requests per minute

Concept: Rate limiting concept

6. Add a health-check route (/health) returning uptime & status

Concept: DevOps readiness

7. Write a test that checks if API returns 404 on unknown route

Concept: Robust testing

■ Outcome: You'll know how to test, structure, and stabilize APIs like a pro.

■ Bonus Challenges

- Add pagination to GET /users (query params: ?page=1&limit;=10)
- Add a search filter (GET /users?name=john)
- Add logging using morgan instead of console
- Add a custom error class for cleaner exceptions
- Deploy on Render or Railway

Week 2: Auth (Password Hashing, Login, Sessions/JWT)

■ Focus

Goal: Learn to authenticate users securely using Express — first with sessions, then with JWTs.

Core Concepts (80/20):

- Password hashing (bcrypt)
- Authentication vs authorization
- Login / logout flow
- Express sessions (stateful auth)
- JSON Web Tokens (stateless auth)
- Protecting routes (middleware)
- Token expiry + refresh pattern
- Basic security hygiene (HTTP-only cookies, .env secrets)

■ SIMPLE LEVEL - Authentication Basics

1. Add user registration route /register that stores {username, password} in memory

Concept: Basic POST

2. Hash password using bcrypt before saving

Concept: Hashing fundamentals

3. Create /login route — verify hashed password using bcrypt.compare()

Concept: Verification

4. On successful login, respond with a simple message (no tokens yet)

Concept: Auth flow

5. Add basic input validation — ensure username + password required

Concept: Input hygiene

6. Return proper HTTP codes: 400 for bad input, 401 for bad credentials

Concept: Error discipline

■ Outcome: You understand how passwords are stored and verified safely.

■ INTERMEDIATE LEVEL - Sessions + JWT Implementation

1. Use express-session to implement session-based login

Concept: Stateful auth

2. Store session in memory (default) — return req.session.user on authenticated route

Concept: Session lifecycle

3. Add a protected route /profile — only accessible if logged in

Concept: Middleware guards

4. Implement logout (req.session.destroy())

Concept: Session invalidation

5. Replace session logic with JWT-based auth using jsonwebtoken

Concept: Stateless auth

6. Add /login that returns a signed JWT (expiresIn: '1h')

Concept: Token issuance

7. Create middleware authenticateToken to verify JWT on protected routes

Concept: Token verification

8. Move JWT secret to .env file

Concept: Environment safety

■ Outcome: You can switch between sessions and JWTs — and know the trade-offs.

■ HARD LEVEL - Advanced Auth Patterns + Testing

1. Add token expiry handling — test expired token returns 401

Concept: Expiration

2. Implement token refresh logic with /refresh-token endpoint

Concept: Refresh flow

3. Store JWT in an HTTP-only cookie (XSS-safe) instead of plain JSON

Concept: Secure cookies

4. Use role-based access (RBAC): e.g. role: 'admin' | 'user'

Concept: RBAC basics

5. Protect admin-only route using role check middleware

Concept: RBAC enforcement

6. Write integration tests with Jest + Supertest for register, login, and protected routes

Concept: Automated testing

7. Add rate-limit to /login using express-rate-limit

Concept: Brute-force defense

■ Outcome: You can build secure, tested, production-grade authentication.

■ Bonus Challenges

- Implement 'Remember Me' option with long-lived JWT
- Add email verification step (mocked email)

- Implement password reset flow (token + expiry)
- Add CORS rules for frontend integration
- Use Redis store for sessions instead of memory

Week 3: Postgres - Schema Design, Migrations, Transactions

■ Focus

Goal: Learn how to design a database schema, manage migrations, and use transactions with Node.js + PostgreSQL.

Core Concepts (80/20):

- Database normalization (1NF, 2NF, 3NF)
- Relationships (one-to-many, many-to-many)
- SQL CRUD queries
- Migrations (schema versioning)
- Transactions (atomic operations)
- Connection pooling
- Query optimization (indexes, limits)

■ SIMPLE LEVEL - PostgreSQL Fundamentals

1. Install PostgreSQL + create a new database user_auth_db

Concept: Setup

2. Use pg npm package to connect Node.js → PostgreSQL

Concept: DB connection

3. Create a table users(id SERIAL PRIMARY KEY, name TEXT, email TEXT UNIQUE, password TEXT)

Concept: Schema creation

4. Write scripts to: insert a user, fetch all users, fetch one by id

Concept: SQL + queries

5. Handle errors (duplicate email, missing values)

Concept: Error handling

6. Add .env for DB credentials (host, user, password, db name)

Concept: Environment safety

7. Use async/await to structure clean DB queries

Concept: Async flow

■ Outcome: You can connect, query, and handle a relational database in Node.

■ INTERMEDIATE LEVEL - Schema Design + Migrations

1. Design a second table posts with columns: id, user_id (FK), title, body, created_at

Concept: Relationships

2. Write SQL for one-to-many relation (user → posts)

Concept: Foreign keys

3. Query all posts with author info using JOIN

Concept: Relational queries

4. Create a folder /migrations and write migration scripts manually (CREATE TABLE, ALTER TABLE, DROP TABLE)

Concept: Schema versioning

5. Automate migrations using node-pg-migrate or knex

Concept: Migration tool

6. Add a rollback migration (DROP TABLE posts)

Concept: Reversibility

7. Integrate migrations in npm run migrate script

Concept: Dev workflow

■ Outcome: You can evolve schema safely and understand database relationships deeply.

■ HARD LEVEL - Transactions + Advanced SQL Operations

1. Implement a function transferBalance(senderId, receiverId, amount) using a transaction

Concept: Atomic operations

2. Simulate an error in the middle — ensure rollback works

Concept: Rollback logic

3. Add a unique constraint + test constraint violation error

Concept: Data integrity

4. Create an index on email and measure performance difference

Concept: Indexing

5. Add pagination (LIMIT, OFFSET) to user list queries

Concept: Performance

6. Write a transaction that inserts a post and updates user's post_count together

Concept: Multi-step transactions

7. Use connection pooling (pg.Pool) to handle multiple requests

Concept: Performance & scalability

■ Outcome: You can handle multi-step database operations safely and efficiently.

■ Bonus Challenges

- Design schema for a small blog: users, posts, comments, likes (with relations)
- Add constraints (ON DELETE CASCADE for posts/comments)
- Add created_at + updated_at with DEFAULT NOW()
- Write a migration that adds a new column to an existing table
- Add indexes on frequently queried columns
- Create a view that joins users + posts for easy querying

Week 4: Integrate DB into App, Add Indexes, Run Benchmarks

■ Focus

Goal: Integrate PostgreSQL with your existing Express app (CRUD + Auth) and understand performance through indexes and basic benchmarks.

Core Concepts (80/20):

- Connection pooling & DB integration
- Query optimization with indexes
- Measuring request latency & DB query time
- Pagination & efficient data retrieval
- Caching basics (conceptual)
- Environment-based config separation (dev/test/prod)

■ SIMPLE LEVEL - Connect Express + Postgres

1. Replace in-memory data from Week 1 + 2 with real PostgreSQL tables (users, posts, etc.)

Concept: Integration

2. Create db.js file that exports a configured pg.Pool instance

Concept: Connection pooling

3. Use async queries inside routes (await pool.query())

Concept: Query execution

4. Update /register, /login, /users, /posts routes to use DB

Concept: Full integration

5. Handle DB errors properly (try / catch, duplicate keys, missing FKs)

Concept: Robustness

6. Move all credentials (DB_HOST, DB_USER, DB_PASS) to .env

Concept: Environment safety

7. Test all routes manually with Postman

Concept: Verification

■ Outcome: Your API now uses PostgreSQL for persistent data storage.

■ INTERMEDIATE LEVEL - Optimize with Indexes + Pagination

1. Use EXPLAIN ANALYZE on queries to see performance

Concept: Query profiling

2. Add indexes on frequently queried fields (email, created_at, etc.)

Concept: Indexing basics

3. Test difference in query time before / after index

Concept: Measurement

4. Add pagination to GET /users and GET /posts (use LIMIT OFFSET)

Concept: Efficiency

5. Implement filtering (GET /users?name=john) and analyze performance

Concept: WHERE clauses

6. Introduce caching idea (mock cache object or Redis later)

Concept: Caching concept

7. Add a DB view user_post_summary joining users + posts

Concept: Pre-computed joins

■ Outcome: You understand how to optimize queries and data access patterns.

■ HARD LEVEL - Benchmarks + Scalability

1. Write a Node script that performs 1000 parallel requests using autocannon or wrk

Concept: Load testing

2. Measure average latency before / after indexing

Concept: Benchmarking

3. Add logging of query duration in middleware

Concept: Performance metrics

4. Use connection pooling properly — compare single connection vs pool

Concept: Pool efficiency

5. Add batch insert endpoint to add 1000 records and measure time

Concept: Bulk operations

6. Implement a small metrics endpoint (/metrics) returning request count, avg time

Concept: Observability

7. Optional: Add Redis caching for hot queries (e.g. /users/top)

Concept: Caching integration

■ Outcome: You can measure and improve your app's performance with real data.

■ Bonus Challenges

- Compare response times with and without index for large dataset
- Try different indexing types (BTREE vs GIN for text search)
- Implement a query log middleware (writes to a logs table)

- Add simple monitoring using morgan + response-time
- Experiment with pool.max settings and observe throughput
- Deploy on a hosted DB (Neon, Supabase, Render Postgres)

Week 5: Background Job System (Redis Queue) + Worker

■ Focus

Goal: Implement a background job system using Redis + Node with a worker that processes jobs asynchronously.

Core Concepts (80/20):

- Background jobs vs synchronous requests
- Redis as a queue (with Bull / BullMQ / Bee-Queue)
- Producers (enqueue job) & Consumers (workers)
- Job retries & failure handling
- Job events (completed, failed, progress)
- Delayed or scheduled jobs
- Graceful shutdown of workers

■ SIMPLE LEVEL - Background Jobs Basics

1. Install Redis locally or use a Docker container

Concept: Queue setup

2. Install Bull (npm i bull) or BullMQ

Concept: Job queue library

3. Create a simple queue emailQueue

Concept: Queue creation

4. Add a route /send-email that adds a job to the queue with {to, subject, body}

Concept: Producer logic

5. Create a worker script that listens to emailQueue and logs the job data

Concept: Consumer logic

6. Test adding multiple jobs manually and watch the worker process them

Concept: Event verification

7. Handle job completion event and log success

Concept: Job events

■ Outcome: You understand how to enqueue jobs and process them asynchronously.

■ INTERMEDIATE LEVEL - Robust Job Processing

1. Add error handling inside the worker — retry failed jobs 3 times

Concept: Reliability

2. Add delayed jobs (e.g., send email 5 minutes later)

Concept: Scheduling

3. Track job progress (job.progress())

Concept: Monitoring

4. Add job-specific data (e.g., userId) and access it in the worker

Concept: Dynamic payloads

5. Log job events: completed, failed, stalled

Concept: Observability

6. Graceful shutdown of worker (queue.close())

Concept: Stability

7. Integrate queue into your Week 2 auth system (send 'Welcome Email' after registration)

Concept: Real-world integration

■ Outcome: You can build reliable background job systems with retries and monitoring.

■ HARD LEVEL - Scaling & Multiple Workers

1. Run multiple workers in parallel and ensure jobs are processed only once

Concept: Concurrency

2. Add job priorities (high/low) and verify order

Concept: Priority queue

3. Implement recurring jobs (e.g., daily report)

Concept: Cron jobs

4. Monitor queue length, failed jobs, and retries (e.g., with bull-board)

Concept: Metrics & dashboard

5. Use Redis connection pooling for multiple queues

Concept: Performance

6. Add a worker that performs CPU-intensive tasks (like image resizing)

Concept: Worker specialization

7. Write integration tests that simulate job creation and worker processing

Concept: Testing async jobs

■ Outcome: You can implement production-ready background processing systems, monitor, and scale them.

■ Bonus Challenges

• Integrate a queue to process large CSV files in batches

- Implement email + notification + logging as separate queues with separate workers
- Add retry delay with exponential backoff for failed jobs
- Persist queue state and job history in Redis for analytics
- Deploy a worker on a separate Node instance (simulate distributed system)

Week 6: Implement CSV Chunking + Batch Processing

■ Focus

Goal: Efficiently process large CSV files using Node, batch database inserts, and integrate with background job systems.

Core Concepts (80/20):

- Streaming vs reading whole file into memory
- CSV parsing with csv-parser or fast-csv
- Processing data in chunks/batches
- Batch inserts into PostgreSQL
- Async handling with Promises / async-await
- Error handling for partial batch failures
- Optional: integrate with background queue for processing

■ SIMPLE LEVEL - CSV Chunking Basics

1. Create a sample CSV file with 100k rows (id,name,email)

Concept: Large dataset simulation

2. Use fs.createReadStream() to read CSV line by line

Concept: Streaming

3. Parse CSV with csv-parser or fast-csv

Concept: CSV parsing

4. Log each row to console without loading entire file into memory

Concept: Memory-efficient processing

5. Count total rows processed using streaming

Concept: Progress tracking

6. Handle basic parsing errors (invalid CSV row)

Concept: Error handling

■ Outcome: You can read very large CSV files efficiently without memory crashes.

■ INTERMEDIATE LEVEL - Batch Processing

1. Collect rows into batches of N (e.g., 500 rows)

Concept: Chunking

2. Insert batch into PostgreSQL using INSERT ... VALUES (...) with multiple rows

Concept: Batch DB inserts

3. Handle batch failures (rollback only failed batch, continue next batch)

Concept: Partial failure handling

4. Log batch progress (Batch 1/200 processed)

Concept: Observability

5. Integrate with background queue (Bull) to process each batch asynchronously

Concept: Queue integration

6. Add command-line script to process CSV file with optional batch size argument

Concept: Configurable batch processing

7. Measure total processing time for 100k+ rows

Concept: Performance awareness

■ Outcome: You can process large CSVs in batches and persist to DB reliably.

■ HARD LEVEL - Large-Scale CSV + Real-World Considerations

1. Process CSV files with 1M+ rows and track memory usage

Concept: Scalability

2. Parallelize batch inserts using Promise.allSettled (careful with DB connections)

Concept: Concurrency

3. Retry failed batches automatically

Concept: Reliability

4. Add data validation per row (e.g., email format, required fields) before DB insert

Concept: Data hygiene

5. Integrate CSV processing as a background job

Concept: Async architecture

6. Implement a progress dashboard (logs, percentage done)

Concept: Observability

7. Optional: support multiple CSV files concurrently using workers

Concept: Scaling

■ Outcome: You can process huge datasets efficiently, in batches, safely, and asynchronously.

■ Bonus Challenges

- Implement CSV streaming with transforms (modify rows before inserting)
- Support different delimiters or headers dynamically
- Integrate with existing auth & users system (import user data)
- Handle CSV errors gracefully and log invalid rows for review

- Optimize batch size dynamically depending on memory usage

Week 7: Add Caching (Redis), TTL Strategies, Basic Invalidation

■ Focus

Goal: Add Redis caching to your Node + Express app, understand TTL strategies, and implement basic cache invalidation.

Core Concepts (80/20):

- Caching layers: Redis as in-memory cache
- Key-value storage
- TTL (time-to-live) for cache expiry
- Cache invalidation strategies (manual, TTL-based, write-through)
- When to cache (hot endpoints)
- Integration with async routes (DB + cache)
- Handling cache misses

■ SIMPLE LEVEL - Redis Caching Basics

1. Install Redis locally or via Docker

Concept: Redis setup

2. Connect Node.js to Redis using ioredis or redis npm package

Concept: Connection

3. Add a simple route /cache-test that stores key: 'foo' → 'bar'

Concept: Set/Get basics

4. Retrieve cached value in another request

Concept: Cache read

5. Set TTL (e.g., 10 seconds) for the key and test expiry

Concept: TTL concept

6. Log cache hits vs misses

Concept: Observability

■ Outcome: You understand how to set, get, and expire values in Redis.

■ INTERMEDIATE LEVEL - Integrate with API Routes

1. Cache GET /users/:id response from PostgreSQL

Concept: Read caching

2. Implement cache check-first: if cache exists return cache, else fetch from DB and set cache

Concept: Cache-aside pattern

3. Add TTL to cached user data (e.g., 60s or 5min)

Concept: Expiry strategies

4. Invalidate cache on updates (PUT /users/:id)

Concept: Basic invalidation

5. Track cache hit/miss metrics with logs

Concept: Observability

6. Cache a collection (e.g., /posts?limit=10&page;=1)

Concept: Pagination caching

7. Add optional cache key versioning (v1:user:123) for safer invalidation

Concept: Key design

■ Outcome: You can integrate Redis caching into real API routes with TTL and invalidation.

■ HARD LEVEL - Advanced Caching Strategies

1. Implement write-through caching: updates write to DB + cache atomically

Concept: Strong consistency

2. Implement cache stampede prevention using locks / mutex

Concept: Performance safety

3. Cache large query results (e.g., top 100 posts) and benchmark speed

Concept: Optimization

4. Add automatic cache invalidation when data changes (e.g., post deleted/updated)

Concept: Event-driven invalidation

5. Implement hot key caching for frequently accessed data

Concept: Performance tuning

6. Explore Redis data structures (hashes, sorted sets) for more complex caching

Concept: Advanced usage

7. Write integration tests for cached endpoints (verify hits/misses, TTL expiry)

Concept: Testing

■ Outcome: You can design production-ready caching layers, improve API performance, and prevent stale or inconsistent data.

■ Bonus Challenges

- Cache batch API responses (multiple users or posts) with a single Redis key
- Combine Redis caching with background job updates
- Experiment with different TTL strategies (short-lived vs long-lived caches)

- Implement a cache dashboard showing hit/miss ratio
- Explore Redis Cluster or Sentinel for distributed caching

Week 8: Rate Limiting + Input Validation + OWASP Checklist

■ Focus

Goal: Make your Node + Express API secure, resilient, and compliant with basic OWASP practices.

Core Concepts (80/20):

- Rate limiting to prevent abuse (DDoS / brute-force)
- Input validation to prevent SQL injection, XSS, etc.
- Basic OWASP Top 10 awareness (injection, broken auth, XSS, security headers)
- Middleware-based security enforcement
- Error handling without leaking sensitive info

■ SIMPLE LEVEL - Rate Limiting & Basic Validation

1. Install express-rate-limit and set up basic global limiter (100 req / 15 min)

Concept: Prevent abuse

2. Add a route-specific limiter for /login (e.g., 5 attempts / 15 min)

Concept: Protect sensitive endpoints

3. Validate incoming requests using express-validator (e.g., email, password length)

Concept: Input sanitation

4. Test invalid inputs and observe 400 responses

Concept: Error handling

5. Add middleware to reject non-JSON requests (Content-Type check)

Concept: Data hygiene

6. Log rate-limited events

Concept: Monitoring

■ Outcome: Your API rejects invalid inputs and prevents brute-force attacks.

■ INTERMEDIATE LEVEL - OWASP Basics & Security Headers

1. Add helmet middleware to set HTTP security headers

Concept: Basic OWASP coverage

2. Sanitize user input to prevent XSS / injection (e.g., express-validator, escaping)

Concept: XSS / injection protection

3. Implement centralized error handler — never leak stack traces in prod

Concept: Security hygiene

4. Enforce HTTPS redirect (conceptual / local dev)

Concept: Transport security

5. Add CORS middleware with restricted origins

Concept: API access control

6. Test common attack patterns manually (SQLi, XSS)

Concept: Security validation

■ Outcome: Your API follows basic OWASP principles and rejects unsafe requests.

■ HARD LEVEL - Advanced Validation + Rate Limiting

1. Implement dynamic rate limits based on IP or user role

Concept: Advanced throttling

2. Validate nested object payloads (e.g., bulk CSV data)

Concept: Complex validation

3. Add token-based request limits (JWT + per-user rate)

Concept: Auth-aware throttling

4. Perform automated security tests with nsp or snyk

Concept: Dependency security

5. Audit API against OWASP Top 10 checklist

Concept: Security compliance

6. Write integration tests for rate limiting and validation

Concept: Automated coverage

7. Optional: integrate CSRF protection for session-based auth

Concept: Stateful security

■ Outcome: Your API is robust, validated, and follows common security best practices.

Week 9: System Design - DB Replication, Read Replicas, Load Balancer

■ Focus

Goal: Learn key patterns for scalable backend systems: database replication, read replicas, and load balancing.

Core Concepts (80/20):

- Database replication (master-slave / primary-replica)
- Read replicas for scaling reads
- Load balancer patterns (round-robin, sticky sessions)
- Horizontal vs vertical scaling
- Connection pooling awareness
- Basics of high availability and fault tolerance

■ SIMPLE LEVEL - Conceptual Setup

1. Draw a diagram of a system with primary DB + one read replica + web servers

Concept: Visualize architecture

2. Configure PostgreSQL primary-replica locally (or Docker)

Concept: Basic replication setup

3. Observe replication logs as data is inserted in primary

Concept: Replication behavior

4. Add a load balancer (nginx or http-proxy) to route requests to multiple Node servers

Concept: Basic load distribution

5. Test read requests on replica, write requests on primary

Concept: Read-write separation

■ Outcome: You understand DB replication + read scaling + load balancing at a conceptual and practical level.

■ INTERMEDIATE LEVEL - Scaling & Monitoring

1. Add multiple read replicas and route queries round-robin

Concept: Horizontal read scaling

2. Use environment config to separate primary vs replica connections

Concept: Connection awareness

3. Implement sticky sessions for login-based requests

Concept: Load balancer patterns

4. Measure response times for primary vs replica queries

Concept: Benchmarking

5. Simulate primary failure and observe failover behavior

Concept: Fault tolerance

6. Add monitoring logs for read/write requests

Concept: Observability

■ Outcome: You can scale reads, balance load, and understand failover basics.

■ HARD LEVEL - Advanced System Design Concepts

1. Implement replica lag detection in Node app (avoid stale reads)

Concept: Consistency awareness

2. Add write sharding logic for very large datasets

Concept: Horizontal writes

3. Implement dynamic load balancing based on server load

Concept: Adaptive scaling

4. Benchmark system under concurrent read/write load

Concept: Performance testing

5. Design caching + read replicas combo to reduce DB load

Concept: Hybrid optimization

6. Document system architecture including failure scenarios

Concept: Design communication

7. Optional: simulate auto-scaling Node app instances behind load balancer

Concept: Scalability practice

■ Outcome: You understand how to scale backend systems while maintaining performance and availability.

Roadmap Summary

This 9-week intensive backend development roadmap follows the Pareto Principle (80/20 rule), focusing on the 20% of concepts that deliver 80% of real-world backend development capability. Each week builds progressively on the previous one, moving from foundational HTTP concepts to advanced system design patterns. The exercises are structured in three difficulty levels: Simple, Intermediate, and Hard, allowing you to gradually build competence and confidence. By completing this roadmap, you will have hands-on experience with: • Building RESTful APIs with Express • Implementing secure authentication systems • Working with PostgreSQL databases • Optimizing performance with indexing and caching • Processing large datasets with background jobs • Implementing security best practices • Designing scalable system architectures The roadmap is designed to be practical and project-based, with each exercise reinforcing core concepts through real-world application. You're encouraged to build one cohesive application throughout the 9 weeks, integrating each new concept as you learn it. Good luck on your backend development journey!