NodeJs Project

Consultant Guide

Node.js Backend Development Course Plan

Learning Resource: Node JS Full Course 2024 | Complete Backend Development Course | Part 1

Day 1: Node.js Fundamentals & Express

Morning – Core Node.js Concepts

- What is Node.js and why use it for backend?
- Installing Node.js and npm
- Understanding the module system
- Using built-in modules: *fs*, *http*, *path*
- Creating a basic server with *http*

Afternoon – Express.js Basics

- Installing Express
- Creating routes and middleware
- Handling GET, POST, PUT, DELETE requests
- Using Postman to test APIs
- Serving static files

Day 2: MongoDB Integration & Project Setup

Morning – Database Integration

- Introduction to MongoDB and Mongoose
- Connecting Node.js to MongoDB
- Defining schemas and models
- CRUD operations with MongoDB
- Error handling and validation

Mini Bank API Project

Tech: Node.js + TypeScript + Express + MySQL + Redis (sessions)

Quality: Unit tests + SonarQube quality gate (no major defects/vulnerabilities)

Testing: Postman collection (required)

High-Level Objective

Build a small banking-style backend service with realistic engineering practices:

- MySQL database integration (FDM credentials)
- Clean architecture (routes/controllers/services/repositories)
- Standardized error handling & structured logging
- Automated unit tests with coverage targets

- SonarQube analysis with quality gate enforcement

Roles (Suggested Split)

- **Dev A:** Accounts & customer/account profile workflows
- **Dev B:** Transactions, transfers, authentication & Redis sessions
- **Shared:** Architecture, error/log standard, shared utilities, code review, documentation, quality gate compliance

Global Constraints (Apply Across Project)

1. **MySQL is the primary data store**
    a. Must use FDM-provided DB credentials (stored only in env vars)
    b. Schema + migrations required
    c. No "in-memory dummy DB" allowed
2. **Redis required for session management** (Epic 4)
3. **Postman collection required** and must cover:
    a. Base flows + protected flows (after Epic 4)
4. **Security & secrets**
    a. Credentials must not be committed
    b. `.env` local only; use `.env.example` for documentation

Below are the required entities:

1. **users** (for authentication)
2. **customers**
3. **accounts**
4. **transactions**
5. **transfers**
6. **sessions (Redis)** *– not stored in MySQL, but included for context*

1) users

Used for login & protecting sensitive endpoints.

Purpose

Represents system users (coaches, admins, or simulated bank employees—not customers).

Table: `users`

Attribute	Type	Null	Description
`id`	BIGINT UNSIGNED (PK, AUTO)	NOT NULL	Unique user ID
`username`	VARCHAR(100)	NOT NULL, UNIQUE	Login username
`password_hash`	VARCHAR(255)	NOT NULL	Hashed password (bcrypt/argon2)
`role`	ENUM('ADMIN', 'STANDARD')	NOT NULL	User role (optional RBAC)
`created_at`	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP	Creation timestamp
`updated_at`	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	

2) customers

Represents bank customers who own accounts.

Table: `customers`

Attribute	Type	Null	Description
`id`	BIGINT UNSIGNED (PK, AUTO)	NOT NULL	Customer ID
`first_name`	VARCHAR(100)	NOT NULL	
`last_name`	VARCHAR(100)	NOT NULL	
`email`	VARCHAR(150)	NOT NULL, UNIQUE	
`phone`	VARCHAR(30)	NULL	
`created_at`	TIMESTAMP	NOT NULL	
`updated_at`	TIMESTAMP	NOT NULL	

Relationships

- A customer **can have many accounts**.

3) accounts

Purpose

A financial account owned by a customer.

Table: `accounts`

Attribute	Type	Null	Description
`id`	BIGINT UNSIGNED (PK, AUTO)	NOT NULL	Account ID
`customer_id`	BIGINT UNSIGNED (FK → customers.id)	NOT NULL	Owner
`type`	ENUM('CHECKING', 'SAVINGS')	NOT NULL	Account type
`currency`	VARCHAR(3)	NOT NULL	ISO currency code (e.g., CAD)
`nickname`	VARCHAR(100)	NULL	User-friendly name
`status`	ENUM('ACTIVE', 'CLOSED')	NOT NULL, DEFAULT 'ACTIVE'	Account lifecycle
`balance`	DECIMAL(12,2)	NOT NULL, DEFAULT 0.00	Current balance (**optional** if computed from transactions)
`created_at`	TIMESTAMP	NOT NULL	
`updated_at`	TIMESTAMP	NOT NULL	

Notes

- You can choose either approach:
  - **Computed balance** from transactions
  - **Stored balance** updated inside MySQL transactions (preferred for transfers)

This design includes a **stored balance**, since transfers require atomic updates.

4) transactions

Purpose

Record every financial event: deposits, withdrawals, and transfer-ledger entries.

Table: `transactions`

Attribute	Type	Null	Description
`id`	BIGINT UNSIGNED (PK, AUTO)	NOT NULL	Transaction ID
`account_id`	BIGINT UNSIGNED (FK → accounts.id)	NOT NULL	Account affected
`type`	ENUM('DEBIT', 'CREDIT')	NOT NULL	Money flow
`amount`	DECIMAL(12,2)	NOT NULL	Positive amount only
`description`	VARCHAR(255)	NULL	Human readable
`category`	VARCHAR(100)	NULL	Optional (e.g., FOOD, TRANSFER, BILL)
`related_transfer_id`	BIGINT UNSIGNED (FK → transfers.id)	NULL	Links back when part of a transfer
`created_at`	TIMESTAMP	NOT NULL	

Rules

- `DEBIT` means money leaving an account
- `CREDIT` means money entering the account
- Amount is always positive; type determines direction

Relationship

- Maybe part of a transfer (not required)

5) transfers

Purpose

Record a transfer event between accounts and link to the two ledger entries in `transactions`.

Table: `transfers`

Attribute	Type	Null	Description
`id`	BIGINT UNSIGNED (PK, AUTO)	NOT NULL	Transfer ID
`from_account_id`	BIGINT UNSIGNED (FK → accounts.id)	NOT NULL	Source
`to_account_id`	BIGINT UNSIGNED (FK → accounts.id)	NOT NULL	Destination

`amount`	DECIMAL(12,2)	NOT NULL	
`memo`	VARCHAR(255)	NULL	Optional direction note
`created_at`	TIMESTAMP	NOT NULL	

How transfers link to transactions

Transfers create two rows in `transactions`:

- One **DEBIT** on `from_account_id`
- One **CREDIT** on `to_account_id`

Each of them will reference:

```
transactions.related_transfer_id = transfers.id
```

This makes it easy to correlate both sides.

6) sessions (Redis)

Purpose

Store sessions for authenticated users.

Stored in Redis (NOT MySQL)

Example structure:

```
session:<sessionId> = {
  userId: 42,
  username: 'john',
  role: 'ADMIN',
  createdAt: 169123456789,
  expiresAt: 169123816789
}
```

TTL is enforced automatically by Redis.

EPIC 1 — Platform Foundation + Accounts (MySQL-backed)

Goal

Set up the service foundation and implement core Accounts APIs using **MySQL persistence** with migrations and seed data.

Endpoints (Minimum)

Health

- `GET /health`

Accounts

- `POST /accounts`
- `GET /accounts?customerId=...`
- `GET /accounts/:accountId`
- `PUT /accounts/:accountId`
- `POST /accounts/:accountId/close`

Acceptance Criteria

Functional (Accounts)

- `POST /accounts` creates an account record in MySQL
  Required fields: `customerId`, `type` (CHECKING|SAVINGS), `currency` (e.g., CAD/USD), optional `nickname`
  Response includes: `accountId`, `status` (ACTIVE), timestamps
- `GET /accounts?customerId=...` returns only accounts belonging to that customer
  If customer has none → return empty list with 200
- `GET /accounts/:accountId` returns account details; if not found → `404`
- `PUT /accounts/:accountId` updates allowed fields only (e.g., nickname/status)
  Invalid updates → `400`
- `POST /accounts/:accountId/close` closes the account
  - If already closed → idempotent response (200 with status CLOSED)
  - If not found → `404`

    - (Optional rule) If balance != 0 → reject with `409` (document your choice)

MySQL Requirements

- Schema created via migrations
- DB access implemented using a proper library/ORM (examples: Prisma, TypeORM, Knex, Sequelize — your choice)
- Seed script exists (at least 1–2 customers)

EPIC 2 — Transactions (Ledger) + Derived Balance

Goal

Implement transactions with MySQL persistence, including pagination/filtering and a consistent ledger approach.

Endpoints (Minimum)

Transactions

- `GET /transactions?accountId=...&limit=...&offset=...&type=...&from=...&to=...`
- `GET /transactions/:transactionId`
- `POST /transactions`

Account summary (recommended)

- `GET /accounts/:accountId/summary` (returns balance + recent activity)

Acceptance Criteria

Functional

- `POST /transactions`
    - Required: `accountId`, `amount`, `type` (DEBIT|CREDIT), `description`
    - Optional: `category`

    - Stores transaction in MySQL
    - If account not found → `404`
    - If invalid amount/type → `400`
- Balance rule:
    - Either compute balance from transactions or store balance and update it transactionally
    - Must be documented in README
    - If enforcing “no overdraft,” a DEBIT that exceeds available funds returns `409`
- `GET /transactions`
    - Requires `accountId`
    - Supports pagination (`limit`/`offset`)
    - Supports filters: `type`, `from`, `to`
    - Deterministic ordering (newest first)
- `GET /transactions/:transactionId`
    - Returns a transaction if exists; else `404`

Data Integrity (MySQL)

- Ensure referential integrity (FKs or application checks)
- Amount stored safely (DECIMAL) not float

EPIC 3 — Transfers (Atomic Operations) + Consistency

Goal

Implement transfers between two accounts with proper transactional behavior in MySQL.

Endpoints (Minimum)

- `POST /transfers`
- `GET /transfers?accountId=...&limit=...&offset=...`
- (Optional) `GET /transfers/:transferId`

Acceptance Criteria

Functional

- `POST /transfers` requires:
  - `fromAccountId`, `toAccountId`, `amount`, optional `memo`
- Validations:
  - Both accounts exist
  - Both are ACTIVE
  - fromAccount ≠ toAccount
  - Sufficient funds (if overdraft not allowed)
- Must create:
  - A transfer record in `transfers`
  - Two transaction records:
    - DEBIT on fromAccount
    - CREDIT on toAccount

Atomicity (MySQL Transaction Required)

- The transfer must be executed inside a **single DB transaction**
- If any insert/update fails → nothing is persisted (no partial state)
- This must be demonstrable via a controlled failure case

EPIC 4 — Authentication + Redis Session Management + Protected Routes

Goal

Add login/logout and Redis-backed sessions. Protect key endpoints and enforce session TTL.

Endpoints (Minimum)

Auth

- `POST /auth/login`

- `POST /auth/logout`
- `GET /auth/me`

Protected Protect at least:

- `POST /accounts`
- `POST /transactions`
- `POST /transfers`

Acceptance Criteria

Functional

- Login:
  - Validates credentials against a user record (seeded in MySQL preferred)
  - Creates a Redis session with TTL
  - Returns session cookie or token referencing session ID
- Protected endpoints:
  - Without session → `401`
  - With valid session → allowed
- Logout:
  - Deletes/invalidate session in Redis
  - After logout, session cannot access protected endpoints
- Session TTL:
  - Must be enforced and demonstrable (short TTL in dev is fine)

Security Requirements

- Don't log secrets (password, tokens, session IDs)
- Validate inputs
- Rate-limiting optional stretch (recommended)

Definition of Done (DoD) — Applies to EVERY Epic / Story

A story is **Done** only when ALL of these are met:

A) Implementation & Architecture

- Follows layered structure:
  - `routes → controllers → services → repositories/data-access`
- Business logic not placed in routes
- TypeScript types/interfaces for request/response payloads
- No hardcoded secrets; uses environment variables

B) MySQL Requirements

- Migrations exist for schema changes
- Seed scripts exist (customers/users sample)
- Uses parameterized queries/ORM safely (prevents SQL injection)
- Monetary amounts stored as **DECIMAL**, not float
- Includes connection pooling/config

C) Validation + Error Handling (MANDATORY)

- Input validation (Zod/Joi/etc.) on all request bodies and query params where relevant
- Central error middleware for consistent error formatting
- Standard error response format:
  - `traceId`, `code`, `message`, `details`
- Correct HTTP codes:
  - 400 validation
  - 401 unauthenticated
  - 403 forbidden (if introduced)
  - 404 not found
  - 409 conflict (e.g., insufficient funds / invalid state)
  - 500 unexpected

D) Logging (MANDATORY)

- Request logging includes:
  - `traceId` (generated or propagated)
  - method, path, status, duration
- Error logs include traceId + contextual info
- No secrets/tokens/passwords in logs

E) Documentation (MANDATORY)

- README updated for each epic:
  - setup instructions
  - env variables explained (`.env.example`)
  - database migration/seed instructions
  - how to run tests
  - how to run sonar scan
- Postman collection updated:
  - includes requests for all endpoints in the epic
  - includes sample payloads & environment variables
  - includes auth flow after Epic 4

F) Testing (Unit Tests)

- Unit tests added for each endpoint behavior:
  - happy path + failure path minimum
- Coverage targets:
  - After Epic 2: **≥ 70%**
  - End of Epic 4: **≥ 80%**
- Tests run in CI and pass

G) SonarQube Quality Gate (MANDATORY)

- Quality gate must pass:
  - **No Major/Critical vulnerabilities**
  - **No Major/Critical bugs**
  - Code smells within acceptable levels (no major)
  - Security hotspots reviewed
- Any new warnings must be fixed before marking done

H) Review Artifacts (PR discipline)

- PR includes:
  - summary
  - how to test
  - screenshot/snippet of:
    - unit test run + coverage

- SonarQube quality gate status