Detailed Node.js API Architecture — Multi-Tenant LMS (Separate DB per Tenant)

This document is a comprehensive, ready-to-implement architecture and implementation guidance for a **Node.js API** that supports a **multi-tenant LMS** with **one database per tenant** (fully isolated data). It includes design decisions, folder structure, code examples (middleware, DB manager, models, controllers, routes), caching, auth, testing, deployment notes, and operational considerations.

Table of contents

34. Operational concerns (backups, migrations, onboarding)
35. Example: tenant onboarding flow
36. Checklist & next steps

---

1. Goals & constraints

- Single codebase serving all tenants (SaaS).
- Each tenant has own DB (MongoDB used in examples).
- No tenant data mixing.
- Fast tenant DB resolution and connection pooling.
- Minimal per-request overhead.
- Support for subdomain-based or header-based tenant identification.
- Horizontal scale via stateless app servers.

2. Design principles

- **Separation of concerns**: controllers call services; services call models.
- **Connection caching**: reuse mongoose connections per tenant.
- **Model factories**: avoid model re-registration errors by generating models for each connection.
- **Fail fast**: if tenant missing or DB unreachable, error early.
- **Minimal middleware logic**: resolve tenant, attach connection & models to `req`.
- **System DB**: small, highly available master DB for tenant metadata and billing.

3. High-level flow

1. HTTP Request arrives (e.g. `GET /courses`).
2. `tenantResolver` middleware identifies tenantId (subdomain or header).
3. `systemDB` lookup gets tenant metadata incl. `dbURI` (cache results in Redis or memory).
4. `dbManager.getConnection(dbURI)` returns cached mongoose connection (or creates one).
5. `modelFactory` binds tenant-specific models to that connection and exposes them on `req.models`.
6. Controller uses `req.models` to fetch/modify tenant data.
7. Response returns only tenant-scoped data.

4. Folder structure (detailed)

```
/lms-multi-tenant
├── src
│   ├── config
│   │   ├── index.js                # environment & constants
│   │   ├── systemDB.js             # connection to system DB
│   │   └── dbManager.js            # tenant DB connection manager
│   │
│   ├── middlewares
```

```
│   │   ├── tenantResolver.js       # identify tenantId & attach tenantInfo
│   │   ├── attachTenantModels.js   # use dbManager to attach models
│   │   └── errorHandler.js         # unified error formatting
│   │
│   ├── tenants                     # tenant-agnostic code (re-usable model
definitions)
│   │   ├── models                  # model definitions (defs only; not bound to
mongoose instance)
│   │   │   ├── user.js
│   │   │   └── course.js
│   │   ├── controllers
│   │   │   ├── authController.js
│   │   │   └── courseController.js
│   │   └── services
│   │       └── courseService.js
│   │
│   ├── system                      # code that runs against system DB
│   │   └── models
│   │       └── tenant.js           # Tenant metadata schema
│   │
│   ├── routes
│   │   ├── publicRoutes.js
│   │   └── tenantRoutes.js
│   │
│   ├── jobs                        # background jobs (bull / agenda)
│   ├── utils
│   ├── app.js
│   └── server.js
├── docker-compose.yml
├── Dockerfile
└── README.md
```

Notes: - `tenants/models/*.js` contain functions that accept a `connection` or `mongoose` instance and return models — this avoids `OverwriteModelError`.

5. System (master) DB design

Collection: tenants

```
{
  "tenantId": "school_01",
  "name": "Green Valley",
  "dbURI": "mongodb://user:pass@host:port/school_01_lms",
  "plan": "standard",
  "subdomain": "gvschool",
```

```
  "status": "active",
  "createdAt": "...",
  "secrets": {
    "jwtSecret": "..."  // optional per-tenant JWT secret
  }
}
```

Cache tenant metadata in Redis or in-memory LRU for quick lookup.

6. Tenant DB design (example schemas)

- `users` — students, teachers, admins
- `courses` — metadata
- `lessons` — content blocks
- `assignments`, `submissions`
- `grades`, `attendance`
- `notifications`

All schema definitions live as *factory functions* that accept a mongoose `connection`.

7. Core modules and responsibilities

tenantResolver (middleware)

- Extract tenant id from subdomain or header
- Validate format
- Lookup tenant metadata in systemDB (cache)
- Attach `req.tenant` = tenantInfo

dbManager (dynamic connections)

- Maintain `Map<dbURI, connection>` cache
- Create new connection if absent
- Reuse existing connections
- Handle connection errors and reconnection strategies

modelFactory

- Provide `getModels(connection)` which returns `{ User, Course, ... }`
- Use `connection.model(name, schema)` instead of mongoose.model

controllers & services

- Controllers are thin: parse request, call service
- Services contains business logic and DB calls using `req.models`

route wiring

- Public routes (health, signup) use systemDB
- Tenant routes use `tenantResolver` + `attachTenantModels`

8. Example code snippets

These examples use **Express** + **Mongoose**. Adapt for other frameworks/databases.

8.1 tenantResolver.js

```
// src/middlewares/tenantResolver.js
const SystemTenant = require('../system/models/tenant');
const tenantCache = new Map(); // replace with Redis in prod

module.exports = async function tenantResolver(req, res, next) {
  try {
    // 1. get tenantId (header or subdomain)
    const tenantId = req.headers['x-tenant-id'] || (req.subdomains &&
req.subdomains[0]);
    if (!tenantId) return res.status(400).json({ error: 'Tenant ID missing' });

    // 2. check cache
    if (tenantCache.has(tenantId)) {
      req.tenant = tenantCache.get(tenantId);
      return next();
    }

    // 3. fetch from system DB
    const tenant = await SystemTenant.findOne({ tenantId }).lean();
    if (!tenant) return res.status(404).json({ error: 'Tenant not found' });

    tenantCache.set(tenantId, tenant);
    req.tenant = tenant;
    next();
  } catch (err) {
    next(err);
  }
};
```

8.2 systemDB.js (connect to system DB)

```
// src/config/systemDB.js
const mongoose = require('mongoose');
const { SYSTEM_DB_URI } = process.env;
```

```
module.exports = async function connectSystemDB() {
  await mongoose.connect(SYSTEM_DB_URI, { useNewUrlParser: true,
useUnifiedTopology: true });
  console.log('Connected to system DB');
};
```

8.3 dbManager.js (tenant connection pooling)

```
// src/config/dbManager.js
const mongoose = require('mongoose');
const CONNECTIONS = new Map(); // key: dbURI, value: { conn, lastUsed }

async function createConnection(dbURI) {
  const conn = await mongoose.createConnection(dbURI, { useNewUrlParser: true,
useUnifiedTopology: true });
  conn.on('error', (e) => console.error('Tenant DB connection error', e));
  return conn;
}

async function getConnection(dbURI) {
  if (CONNECTIONS.has(dbURI)) {
    const entry = CONNECTIONS.get(dbURI);
    entry.lastUsed = Date.now();
    return entry.conn;
  }
  const conn = await createConnection(dbURI);
  CONNECTIONS.set(dbURI, { conn, lastUsed: Date.now() });
  return conn;
}

module.exports = { getConnection };
```

Notes: In production, monitor connection count and disconnect idle ones after TTL.

8.4 model factory (user, course)

```
// src/tenants/models/user.js
module.exports = function createUserModel(connection) {
  const mongoose = require('mongoose');
  const { Schema } = mongoose;
  const schema = new Schema({
    name: String,
    email: { type: String, index: true },
    role: { type: String, enum: ['student','teacher','admin'] },
```

```
    passwordHash: String,
    createdAt: Date
  });
  return connection.model('User', schema);
};

// src/tenants/models/course.js
module.exports = function createCourseModel(connection) {
  const mongoose = require('mongoose');
  const { Schema } = mongoose;
  const schema = new Schema({
    title: String,
    description: String,
    createdBy: { type: Schema.Types.ObjectId, ref: 'User' },
    createdAt: Date
  });
  return connection.model('Course', schema);
};
```

8.5 attachTenantModels middleware

```
// src/middlewares/attachTenantModels.js
const { getConnection } = require('../config/dbManager');
const createUserModel = require('../tenants/models/user');
const createCourseModel = require('../tenants/models/course');

module.exports = async function attachTenantModels(req, res, next) {
  try {
    const tenant = req.tenant;
    if (!tenant) return res.status(400).json({ error: 'Tenant not resolved' });

    const conn = await getConnection(tenant.dbURI);
    // create or reuse models bound to conn
    req.models = {
      User: createUserModel(conn),
      Course: createCourseModel(conn),
    };

    next();
  } catch (err) {
    next(err);
  }
};
```

Note: calling `createUserModel(conn)` repeatedly should be safe because `connection.model('User')` returns existing model if already registered on that connection.

8.6 controller & service example

```
// src/tenants/services/courseService.js
class CourseService {
  constructor(models) { this.Course = models.Course; }

  async listAll() { return this.Course.find().lean(); }
  async create(payload) { return this.Course.create(payload); }
}
module.exports = CourseService;

// src/tenants/controllers/courseController.js
const CourseService = require('../services/courseService');

exports.listCourses = async (req, res, next) => {
  try {
    const svc = new CourseService(req.models);
    const courses = await svc.listAll();
    res.json({ data: courses });
  } catch (err) {
    next(err);
  }
};

exports.createCourse = async (req, res, next) => {
  try {
    const svc = new CourseService(req.models);
    const created = await svc.create(req.body);
    res.status(201).json({ data: created });
  } catch (err) { next(err); }
};
```

8.7 route wiring

```
// src/routes/tenantRoutes.js
const express = require('express');
const tenantResolver = require('../middlewares/tenantResolver');
const attachTenantModels = require('../middlewares/attachTenantModels');
const courseController = require('../tenants/controllers/courseController');

const router = express.Router();
```

```
// Apply tenant middlewares for all tenant routes
router.use(tenantResolver);
router.use(attachTenantModels);

router.get('/courses', courseController.listCourses);
router.post('/courses', courseController.createCourse);

module.exports = router;
```

8.8 app.js

```
// src/app.js
const express = require('express');
const tenantRoutes = require('./routes/tenantRoutes');
const systemDBConnect = require('./config/systemDB');

const app = express();
app.use(express.json());

// system DB connect
systemDBConnect();

app.use('/tenant', tenantRoutes);

app.use(require('./middlewares/errorHandler'));

module.exports = app;
```

9. Authentication & Authorization

- Use **JWT** that includes `tenantId` claim.
- Option A: **Global JWT secret** (faster); verify server-side and check `tenantId` matches `req.tenant`.
- Option B: **Per-tenant JWT secret** (more secure for full isolation) stored in system DB or secret manager. Use this for signing tokens for tenant-specific admin panels.
- Store refresh tokens in tenant DB `RefreshTokens` collection.
- Always check roles in controllers/services.

10. File uploads & media handling

- Use S3 (or equivalent) for media; store path and access control in tenant DB.
- Consider tenant-prefixed keys: `tenantId/uploads/{fileId}`.
- For private media, issue signed URLs with short expiry.

11. Caching & performance

- Cache tenant metadata (system DB) in Redis with TTL.
- Cache frequently read data (course lists) per tenant with keys namespaced by `tenantId`.
- Use Redis for session store or rate limiting.

12. Background jobs & scheduling

- Use BullMQ (Redis) or Agenda (Mongo) for tasks: reports, email digests, grade processing.
- Jobs should store `tenantId` and obtain DB connection inside worker using dbManager.

13. Observability & logging

- Correlate logs by `requestId` and `tenantId`.
- Structured logs (JSON) to send to ELK / Datadog.
- Export metrics: request latency per tenant, DB connection counts, job queue length.

14. Security considerations

- Validate and sanitize all inputs (prevent injection).
- Rate limit per tenant (prevent noisy neighbor).
- Ensure backups and encryption at rest.
- Principle of least privilege for system DB credentials.
- Rotate per-tenant secrets when needed.

15. Testing strategy

- Unit tests for services and controllers (use in-memory Mongo or mocked models).
- Integration tests that spin up test tenants (use ephemeral DBs).
- End-to-end tests for onboarding flow and key tenant actions.

16. CI/CD and deployment

- Dockerize the app. Build image and push to registry.
- Use K8s or ECS to run stateless pods behind LB.
- Use autoscaling based on CPU/RPS.
- Migrations: run per-tenant migrations individually using a migration tool and `tenantId` loop or run migration on connection creation.

17. Operational concerns

- **Backups**: schedule per-tenant DB backups and ensure fast restore for one tenant.
- **Migrations**: store migration state in systemDB per tenant.
- **Onboarding**: create DB, run migrations, seed admin user, add tenant metadata to systemDB.

18. Example: tenant onboarding flow

1. Admin signs up on SaaS portal.
2. System creates DB (provisioning script) — or create when first request arrives.
3. System writes tenant record to `tenants` collection with `dbURI`.
4. Optionally seed default data (courses, roles, sample content).
5. Start issuing tenant-specific JWTs.

19. Checklist & next steps

- [ ] Choose DB (Mongo/Postgres). This doc uses Mongo examples.
- [ ] Implement system DB & tenant registration.
- [ ] Implement tenantResolver and caching.
- [ ] Implement dbManager with connection pooling and TTL eviction.
- [ ] Convert tenant models to factory functions.
- [ ] Implement authentication (JWT) and RBAC.
- [ ] Add logging, monitoring, and backups.
- [ ] Plan migrations strategy.

Appendix: Helpful pointers & gotchas

- **Don't use** `mongoose.model(name, schema)` **globally** — always attach models to a connection.
- **Avoid creating a new model on every request**: keep per-connection model registration cached.
- **Connection limits**: cloud-hosted Mongo's connection limits can be reached — use pooling and limit number of Node workers or shard tenants across clusters.
- **Tenant explosion**: if many tiny tenants cause overhead, consider grouping small tenants into shared DB for a cheaper plan (hybrid approach).

If you'd like, I can now: - Provide the **complete code** for `dbManager` including eviction and reconnection logic.
- Convert the examples to **Postgres & Sequelize** instead of Mongoose.
- Create a **deployment-ready Dockerfile & Kubernetes manifests**.
- Generate **unit test examples** (Jest) for services and controllers.

Tell me which deliverable you want next and I'll prepare it.