ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

Software Engineering and Development

M1: DIA, OCC, CCC
2025-2026

Lab session 7: Connect to MongoDB Atlas and Implement Unit pand Integration Tests

Introduction

In case you did not create your MongoDB Atlas database in the previous session, this lab starts from the very beginning. You will first set up your Atlas account, create your cluster, configure IP access, create your database user, and connect your Node.js backend to your cloud database.

If you already created your cluster earlier, you should still follow these steps to verify your configuration, credentials, and IP access, and ensure that your application prints:

```
Connected to MongoDB: <your-db-name>
```

Only after confirming that your database connection works correctly may you proceed with Mongoose models, controllers, middleware, query helpers, and testing.

Throughout this lab, you must adapt all work to **your own project**, based on the Jira stories, epics, and use cases you designed earlier. The hospital example is provided only as a reference.

Learning Goals

By the end of this lab, you will be able to:

- create a MongoDB Atlas cluster from scratch,
- configure IP access and a database user,
- connect a Node.js backend to Atlas,
- use `mongosh` to inspect your cloud database,
- define Mongoose schemas and models for your application's entities,
- refactor your API using the MVC structure (Model–Controller–Routes),
- implement Mongoose middleware and query helpers,

- run integration tests with Vitest and Supertest,
- apply the full DevOps workflow: branch → commit → PR → CI → merge.

1 MongoDB Atlas Setup

1.1 Account Creation

1. Navigate to `https://www.mongodb.com/atlas`.
2. Create a free account (using Google or your email).

1.2 Cluster Creation

1. Create a cluster ("Build a cluster" button).
2. Select the Free option.
3. Choose your cluster name, example: `mern-project`.
4. Choose a region (EU recommended).
5. Click **Create Cluster**.

1.3 Security Quickstart

On the "Security Quickstart" page, set the following:

1. Choose a username and a password:
   - username: `your-name`
   - password: `your-password`
   - role: Read/Write
2. In the connection section, enable the "Cloud Environment" connection.
3. Click "Finish and Close".

1.4 Connection String

Go to:

Database → Connect → Drivers → Node.js

Copy the URI:

```
mongodb+srv://your-name:<password>@mern-project.abc12.mongodb.net/<
    dbname>?retryWrites=true&w=majority
```

Replace <db_password>with the password you have set previously.

1.5 Environment Variables

Create a `.env` file:

```
MONGO_URI=mongodb+srv://your-name:<password>@mern-project.abc12.
    mongodb.net/<dbname>?retryWrites=true&w=majority
PORT=3000
```

1.6 Database Connection Module

Create `src/db/mongo.js`:

```
import { MongoClient } from "mongodb";
import dotenv from "dotenv";
dotenv.config();

const client = new MongoClient(process.env.MONGO_URI);
let db;

export async function connectToDb() {
  await client.connect();
  db = client.db();
  console.log("Connected to MongoDB:", db.databaseName);
}

export function getDb() {
  return db;
}
```

1.7 Start the Server

```
// src/index.js
import app from "./app.js";
import { connectToDb } from "./db/mongo.js";

const port = process.env.PORT || 3000;

async function start() {
  await connectToDb();
  app.listen(port, () => {
    console.log(`API running at http://localhost:${port}`);
  });
}

start();
```

Your terminal must show:

```
Connected to MongoDB: mernproject
API running at http://localhost:3000
```

2 Inspecting the Database with Mongosh

Install the MongoDB shell:

https://www.mongodb.com/try/download/shell

Connect:

```
mongosh "<your_full_connection_uri>"
```

Useful commands:

```
show dbs
use mernproject
show collections
db.<collection>.find().pretty()
db.<collection>.countDocuments()
```

3 Mongoose Models

Install Mongoose:

```
npm install mongoose
```

Adapt this model to the entities in your project:

```
// src/models/doctor.model.js

import mongoose from "mongoose";

const doctorSchema = new mongoose.Schema(
  {
    name: { type: String, required: true },
    specialty: { type: String, required: true },
    active: { type: Boolean, default: true }
  },
  { timestamps: true }
);

export default mongoose.model("Doctor", doctorSchema, "Doctors");
```

4 MVC Refactoring

The MVC architecture (Model–View–Controller) helps you structure your backend by separating responsibilities: the **Model** defines your data structure and all database interactions using Mongoose, the **Controller** contains the business logic, validates inputs, handles errors, and determines the response, and the **Routes** only map HTTP endpoints to controller functions without containing any logic. In this lab, you must reorganize your project following this structure by creating a Mongoose model for your entity, moving all logic from your previous routes into controller functions, and keeping your route files minimal and limited to endpoint definitions. This separation makes your code easier to maintain, test, and extend as your application evolves. Here is an example.

4.1 Controllers

```
// src/controllers/doctor.controller.js

import Doctor from "../models/doctor.model.js";

export async function listDoctors(req, res, next) {
  try {
    const docs = await Doctor.find().lean();
    res.status(200).json(docs);
  } catch (err) {
    next(err);
  }
}

export async function createDoctor(req, res, next) {
  try {
    const { name, specialty } = req.body;
    const created = await Doctor.create({ name, specialty });
    res.status(201).json(created);
  } catch (e) {
    next(e);
  }
}
```

4.2 Routes

```
// src/routes/doctor.routes.js
import express from "express";
import {
  listDoctors,
  createDoctor
```

```
} from "../controllers/doctor.controller.js";

const router = express.Router();

router.get("/", listDoctors);
router.post("/", createDoctor);

export default router;
```

5 Unit Tests and Integration Tests

In this part, you will write both unit tests and integration tests. A **unit test** focuses on a small piece of code in isolation (for example a pure function or a controller helper) without calling the database or the HTTP layer. An **integration test** exercises several parts of the system together (for example an Express route that uses a controller and a Mongoose model, tested via HTTP requests with Supertest).

Example of a unit test

Consider a small utility function that validates a doctor object:

```
// src/utils/doctorValidation.js
export function isValidDoctor(data) {
  if (!data) return false;
  if (!data.name || typeof data.name !== "string") return false;
  if (!data.specialty || typeof data.specialty !== "string") return
     false;
  return true;
}
```

A unit test verifies this function alone, without Express or MongoDB:

```
// tests/unit/doctorValidation.test.js
import { describe, it, expect } from "vitest";
import { isValidDoctor } from "../../src/utils/doctorValidation.js";

describe("isValidDoctor", () => {
  it("returns true for a valid doctor object", () => {
    const doctor = { name: "Dr. Test", specialty: "Cardiology" };
    expect(isValidDoctor(doctor)).toBe(true);
  });

  it("returns false if name is missing", () => {
    const doctor = { specialty: "Cardiology" };
    expect(isValidDoctor(doctor)).toBe(false);
  });
```

```
  it("returns false if specialty is not a string", () => {
    const doctor = { name: "Dr. Test", specialty: 42 };
    expect(isValidDoctor(doctor)).toBe(false);
  });
});
```

Example of an integration test

An integration test checks a full request through the API, using Express, the controller, the Mongoose model, and MongoDB:

```
// tests/integration/doctorsApi.test.js
import request from "supertest";
import { describe, it, expect, beforeEach } from "vitest";
import app from "../../src/app.js";
import Doctor from "../../src/models/doctor.model.js";

describe("Doctors API (integration)", () => {
  beforeEach(async () => {
    await Doctor.deleteMany({});
  });

  it("creates a doctor and then returns it in the list", async () =>
      {
    // Create a new doctor
    const createRes = await request(app)
      .post("/api/doctors")
      .send({ name: "Dr. Integration", specialty: "Testing" });

    expect(createRes.status).toBe(201);
    expect(createRes.body).toHaveProperty("_id");

    // List all doctors
    const listRes = await request(app).get("/api/doctors");

    expect(listRes.status).toBe(200);
    expect(Array.isArray(listRes.body)).toBe(true);
    expect(listRes.body.length).toBe(1);
    expect(listRes.body[0].name).toBe("Dr. Integration");
  });
});
```

Implement unit tests and integration tests similar to the examples above, adapted to the entities and logic of your own project.