Modex Assessment
Healthcare Lab Test Slot Booking – Backend

1. PROJECT OVERVIEW

This project is a **Healthcare Lab Test Slot Booking Backend System**.
It allows:

- Admin to create lab test slots
- Public users to view available slots
- Public users to book a slot (with availability validation)
- Fetch a single booking by ID
- Automatically track:
    - Total capacity
    - Confirmed bookings
    - Remaining seats

The system is fully powered by:

- **Node.js + Express.js**
- **PostgreSQL (pg module)**
- **Thunder Client for testing APIs**

This backend forms the foundation for any healthcare lab booking platform, enabling secure creation, listing, and booking of lab tests.

2. FEATURES IMPLEMENTED

a. Create Lab Test Slot (Admin API)

Admin can create a new test slot with:

- Test name
- Start time
- Total capacity

POST /admin/slots

b. List All Slots with Remaining Capacity

Returns every slot with:

- Total capacity
- Confirmed bookings
- Remaining seats

GET /slots

c. Book a Slot

A patient can book 1 or more seats if available.

POST /slots/:id/book

Validates:

- Slot exists
- Remaining capacity >= requested seats
- Reduces available capacity automatically.

d. Get Booking Details

Fetch details of a specific booking.

GET /bookings/:id

e. Health Check Route

Checks if the server and database are connected.

GET /health

3. SIMPLE STEPS ON HOW TO EXECUTE THIS ASSIGNMENT

Step 1 : Open File Explorer and very ever u want to create u can create a File name Called "**Modex-Healthcare-Booking"**

Step 2: Double click on "**Modex-Healthcare-Booking**"
Step 3: Now create a New Folder Called "**backend"**
Step 4: Open VS Code

- Go to on File on top most one
- Click on Open Folder
- Now Open "**Modex-Healthcare-Booking"**
- Now click on that "**backend"** and press enter and u can see select folder press on it

Now our Present Project Structure Loots like:

Now your structure on Desktop looks like:

- Desktop
  └─ 📁 Modex-Healthcare-Booking
    └─ 📁 backend

Step 5: Open Terminal where u can see it at the top corner click on it

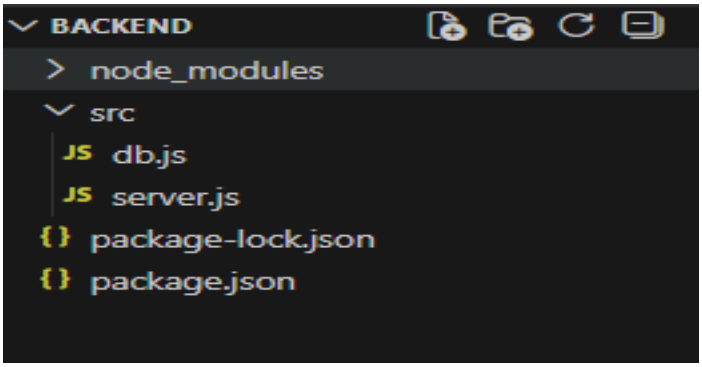
Step 6: Paste this Command: "**npm init -y"** (This creates a file called **package.json)**

Step 7: "npm install express pg dotenv cors"

Step 8: "npm install --save-dev nodemon"

Step 9: Create the **"src"** folder down to Backend

Step 10: Now under src Create a new files called: "**server.js" & "db.js"**

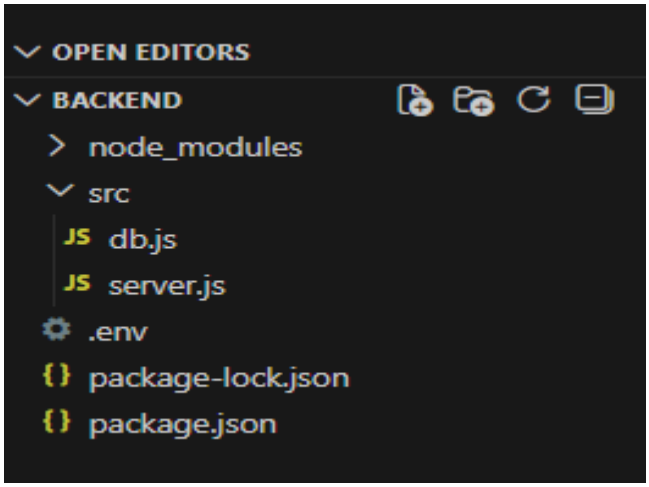


Step 11: Now Update the code in **“package.json”** and Save it (CTRL+S)

Code:

```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "src/server.js",
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^3.1.11"
  },
  "dependencies": {
    "cors": "^2.8.5",
    "dotenv": "^17.2.3",
    "express": "^5.2.1",
    "pg": "^8.16.3"
  }
}
```

Step 12: Click on Backend bestie to that there a symbol to create a New File click on it and create "**.env"** file



Step 13: Open "**src/db.js"** and add the code and save it.

Code:

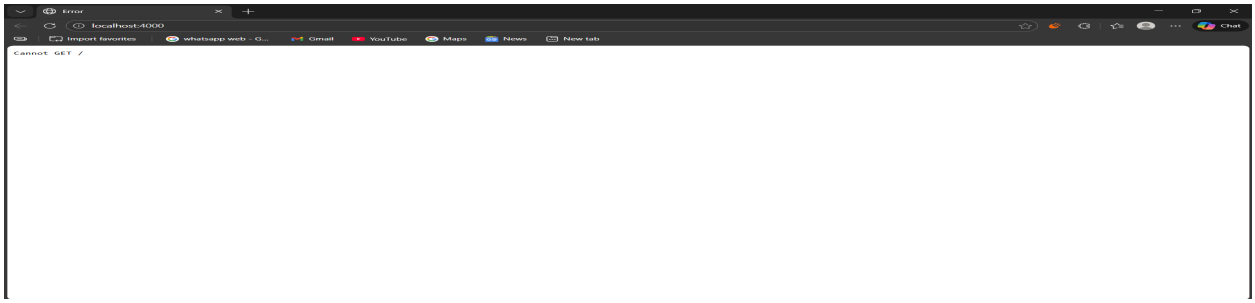
```
const { Pool } = require("pg");

require("dotenv").config();

// Create a connection pool using the DATABASE_URL from .env

const pool = new Pool({

 connectionString: process.env.DATABASE_URL,

 ssl: process.env.DB_SSL === "true" ? { rejectUnauthorized: false } : false

});

// Helpful logging to know if database connects

pool.connect()

 .then(() => console.log("✅ Connected to PostgreSQL Database"))

 .catch(err => console.error("❌ Database connection failed:", err.message));

module.exports = { pool };
```

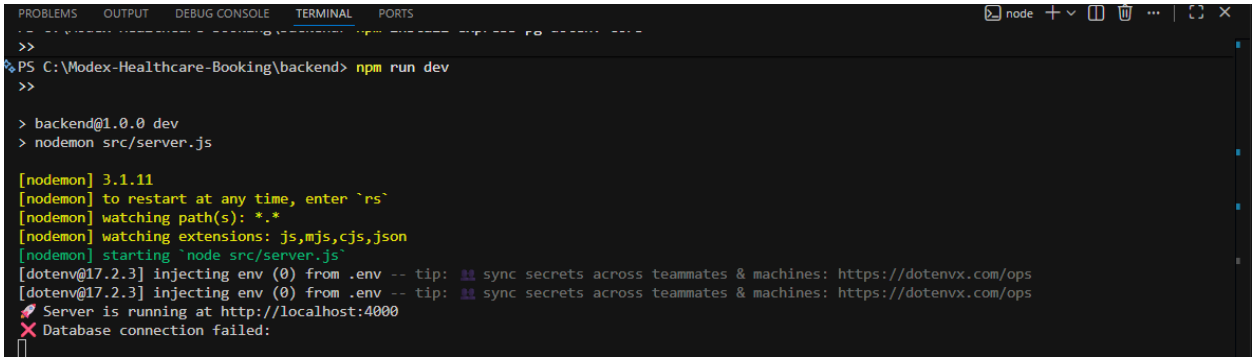
Step 14: Open "**src/server.js**" and add the code and save it.

Code:

```
const express = require("express");

const cors = require("cors");

require("dotenv").config();

const { pool } = require("./db");

const app = express();

// Middlewares

app.use(cors());

app.use(express.json());

// Test route - will tell us if server is running

app.get("/health", async (req, res) => {

 try {

  await pool.query("SELECT 1"); // check db is reachable

  res.json({ status: "ok", message: "Server and DB working fine 🚀" });

 } catch (error) {

  res.json({ status: "ok", message: "Server running, DB not connected yet ❗ " });

 }

});

const PORT = process.env.PORT || 4000;

app.listen(PORT, () => {

 console.log(`🚀 Server is running at http://localhost:${PORT}`);

});
```

Step 15: In VS Code Terminal Type this "**npm run dev**"

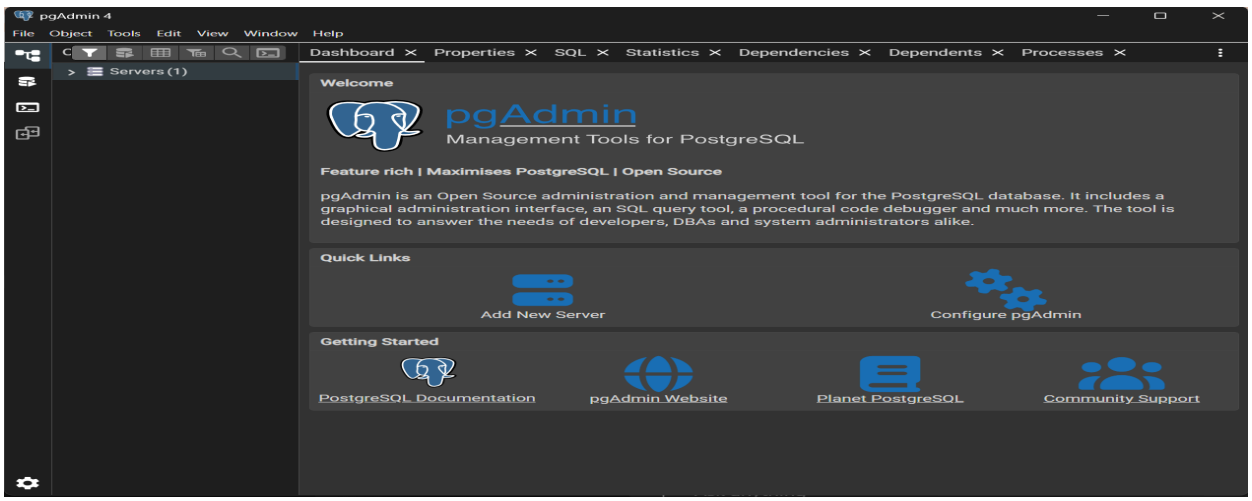




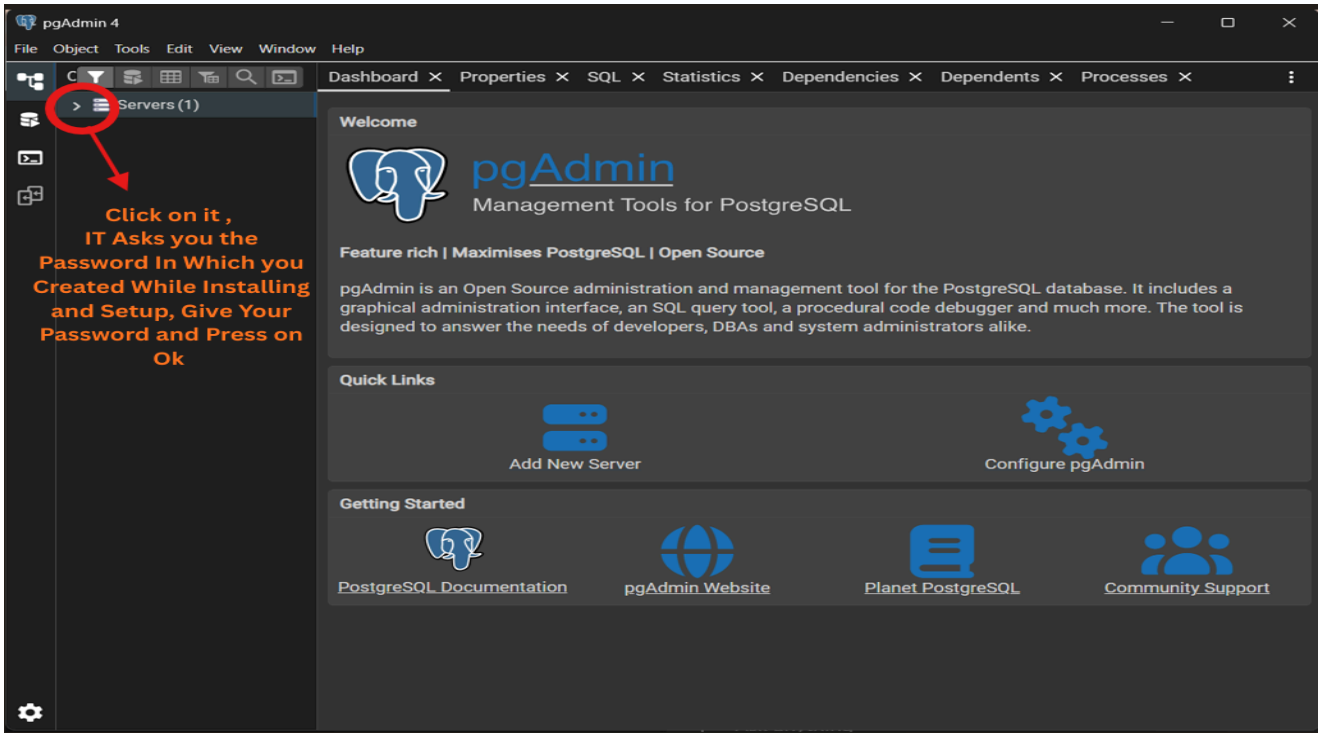
Step 16: Install PostgreSQL and setup it

Step 17: After installation:

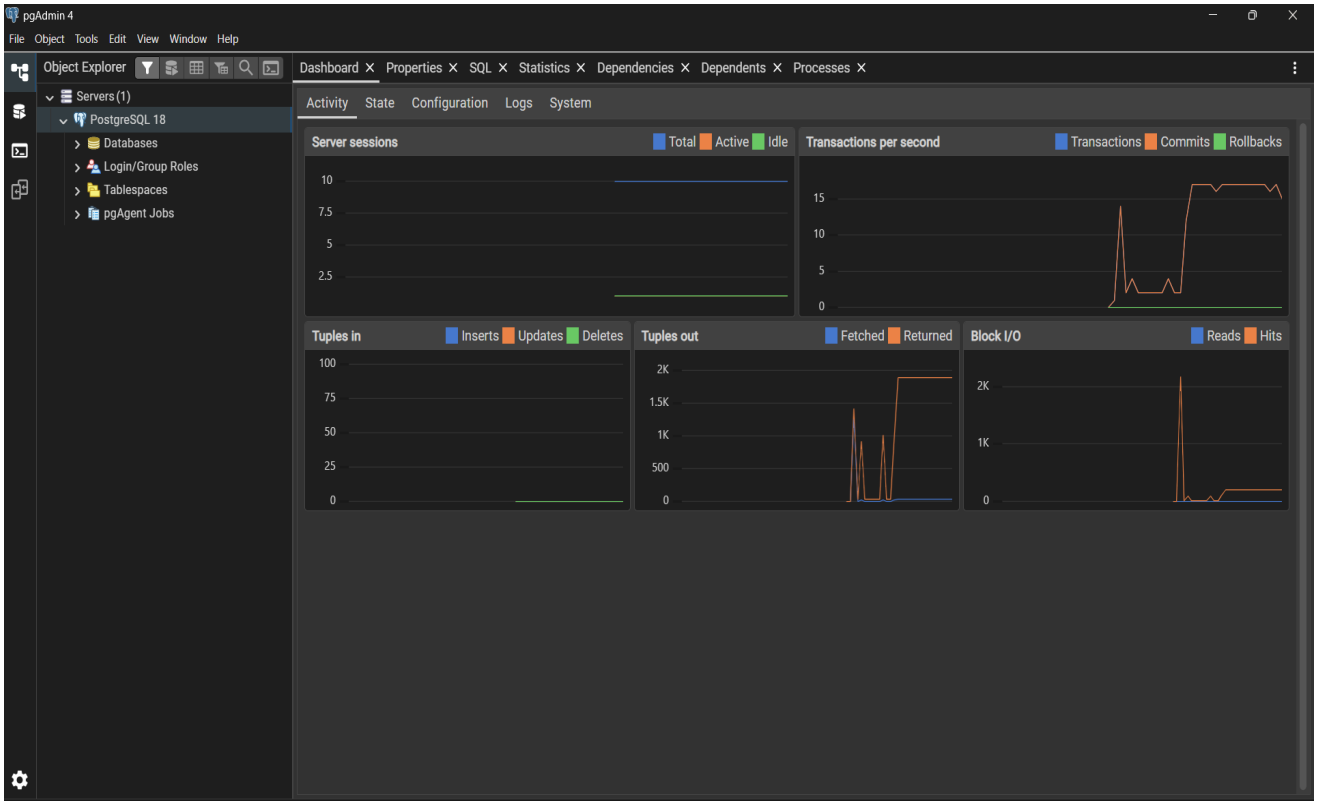
1. Press **Windows Key**
2. Type **pgAdmin**
3. Click **pgAdmin 4**



Step 18: Follow the Step in Image.

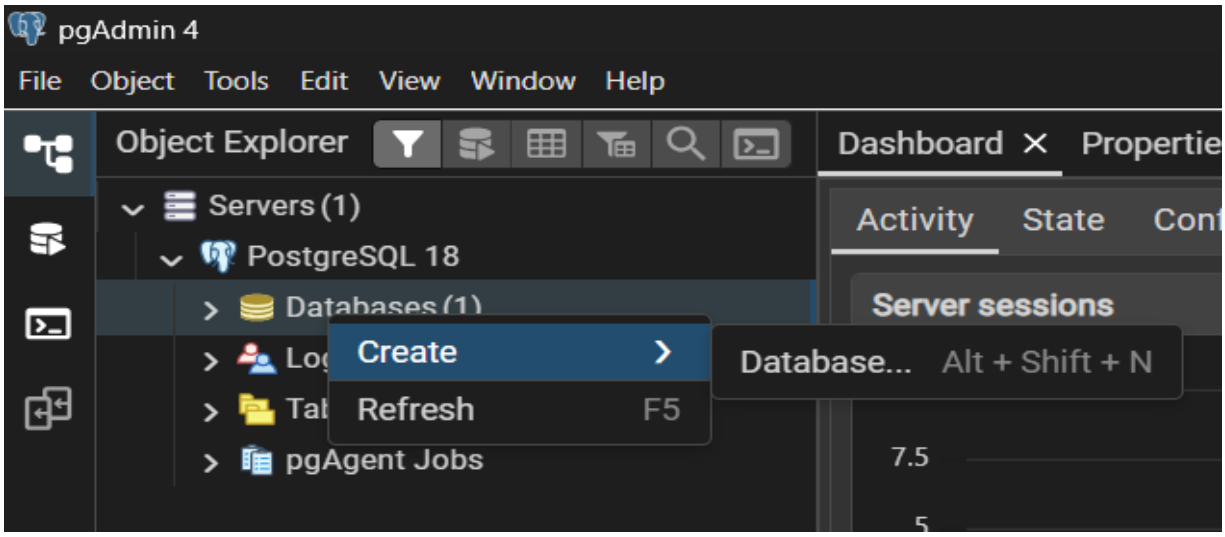


Step 19: We can see How it looks after giving Password in the Below Image.

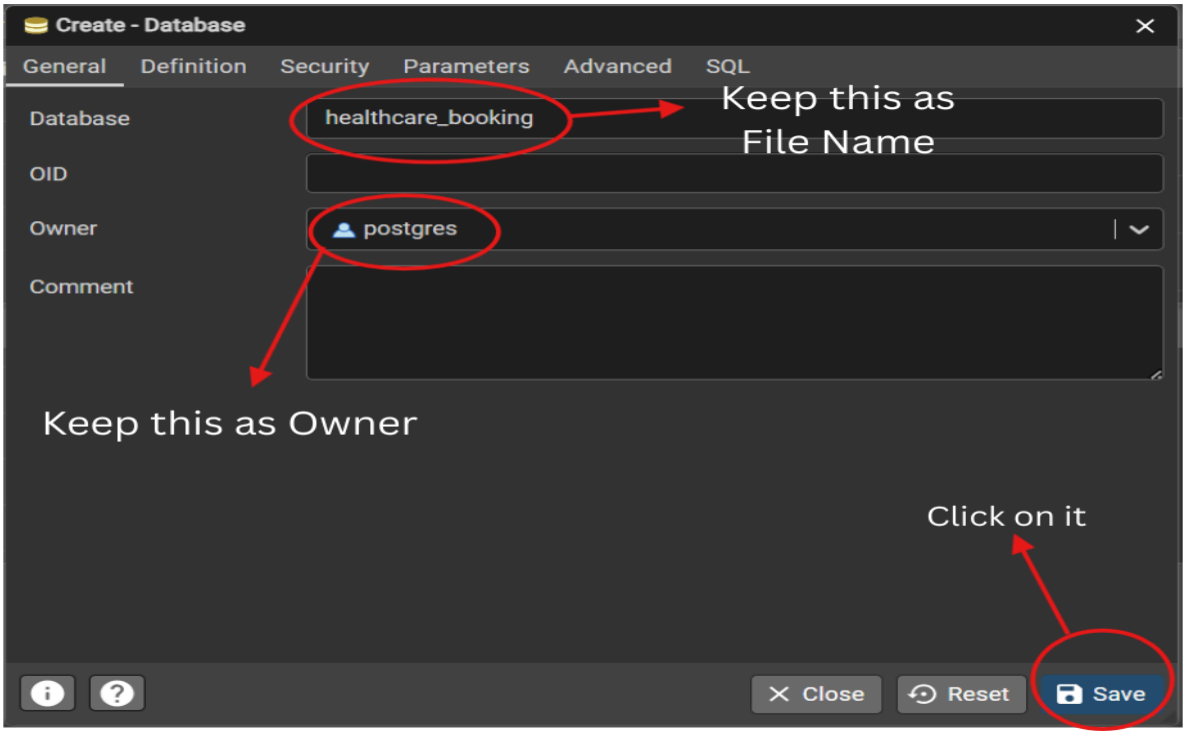


Step 20: Now Creating a New Database

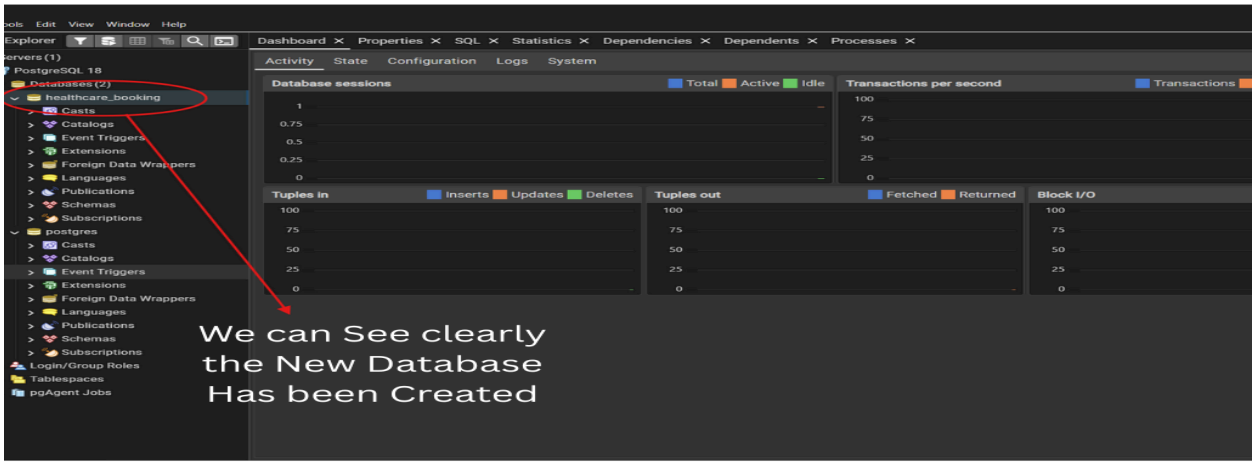
1. On the left sidebar, expand:
2. Right-click **Databases**
3. Click **Create** → New Database as **"healthcare_booking"**



Step 20: Follow the below image



Step 21: Now We had Successfully Created a New Database



Step 22: Update .env so backend can connect to PostgreSQL

Paste this Code in .env:

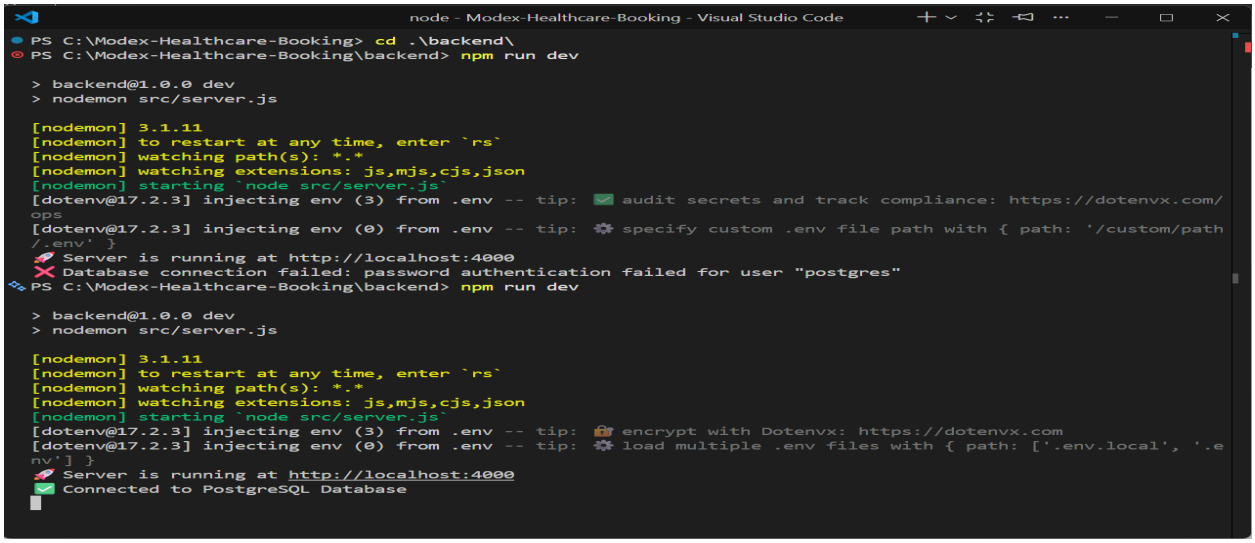
PORT=4000

DATABASE_URL=postgres://postgres:Vardhan%40123@localhost:5432/healthcare_booking

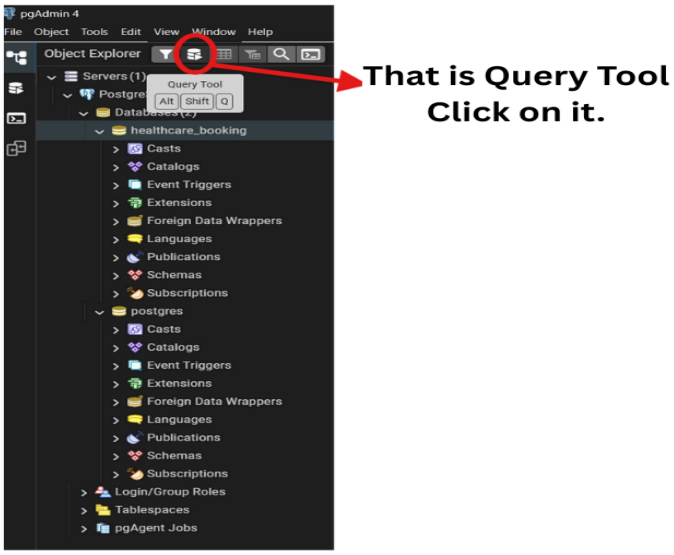
DB_SSL=false

And save it.

Step 23: Now run this command in VS Code terminal: **“npm run dev”**



Step 24: Go to PostgreSQL and click on the Query Tool

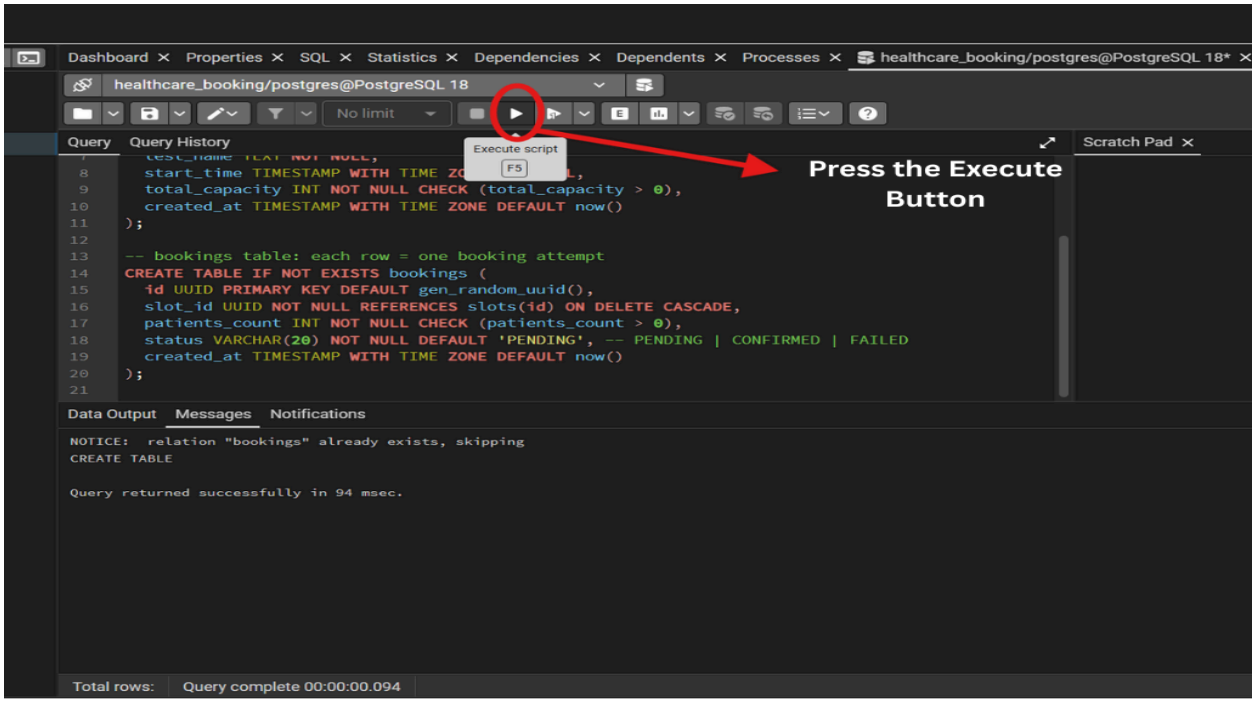


Step 25: In Query Tool Type the Code

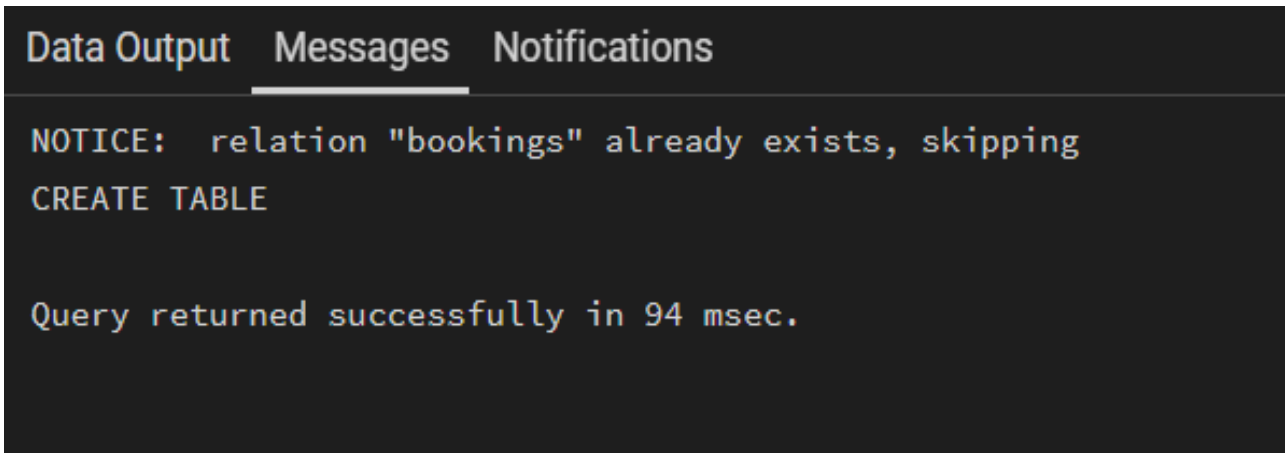
Code:

```
-- enable pgcrypto so we can use gen_random_uuid() for UUID defaults
CREATE EXTENSION IF NOT EXISTS pgcrypto;
-- slots table: each row = one lab test slot
CREATE TABLE IF NOT EXISTS slots (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  test_name TEXT NOT NULL,
  start_time TIMESTAMP WITH TIME ZONE NOT NULL,
  total_capacity INT NOT NULL CHECK (total_capacity > 0),
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);
-- bookings table: each row = one booking attempt
CREATE TABLE IF NOT EXISTS bookings (
```

```
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  slot_id UUID NOT NULL REFERENCES slots(id) ON DELETE CASCADE,
  patients_count INT NOT NULL CHECK (patients_count > 0),
  status VARCHAR(20) NOT NULL DEFAULT 'PENDING', -- PENDING | CONFIRMED | FAILED
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);
```



Step 26: The Output will be like the Image Below



Step 27: Check tables exist, Open Query Tool and Type the SQL Query in it and Click on Execute then we can know that the Table has been Created or not.

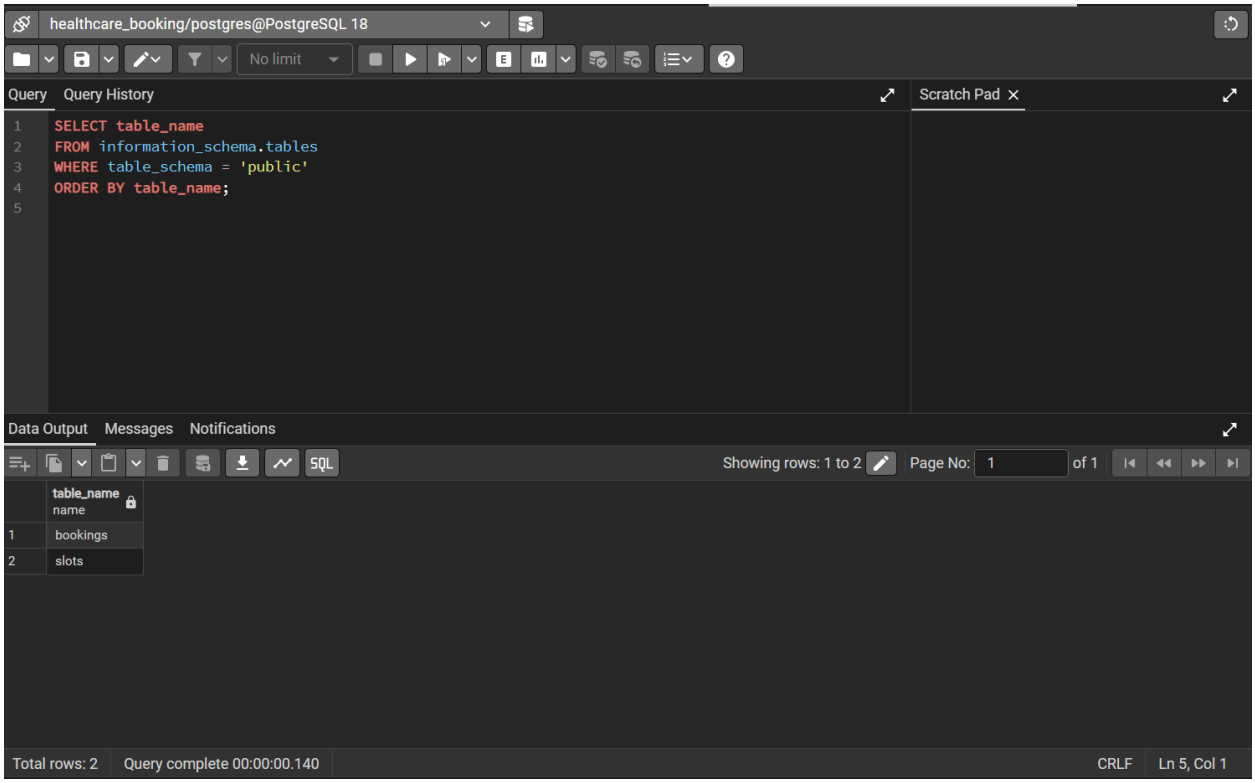
SQL Query:

SELECT table_name

FROM information_schema.tables

WHERE table_schema = 'public'

ORDER BY table_name;



Step 28: In VS Code under src/server.js Enter the New Code.

Code:

const express = require("express");

const cors = require("cors");

require("dotenv").config();

const { pool } = require("./db");

```
const app = express();

// Middlewares

app.use(cors());

app.use(express.json());

// Default route (homepage)

app.get("/", (req, res) => {

 res.send("🧪 Healthcare Lab Test Slot Booking Backend is running successfully!");

});

// Health check route

app.get("/health", async (req, res) => {

 try {

  await pool.query("SELECT 1"); // test DB connection

  res.json({

   status: "ok",

   message: "Server is running and database connection is successful 🚀"

  });

 } catch (error) {

  res.json({

   status: "ok",

   message: "Server is running, but database is not connected yet ❗ ",

   error: error.message

  });

 }

});

// ================================================

// 🔵 ADMIN: CREATE A NEW LAB TEST SLOT
```

```
// ================================================

app.post("/admin/slots", async (req, res) => {
  try {
    const { test_name, start_time, total_capacity } = req.body;


    if (!test_name || !start_time || !total_capacity) {
      return res.status(400).json({ error: "All fields are required." });
    }
    const result = await pool.query(
      `INSERT INTO slots (test_name, start_time, total_capacity)
       VALUES ($1, $2, $3)
       RETURNING *`,
      [test_name, start_time, total_capacity]
    );
    res.json({
      message: "Slot created successfully!",
      slot: result.rows[0]
    });
  } catch (error) {
    console.error("Create Slot Error:", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
});
// ================================================

// START SERVER

// ================================================
```

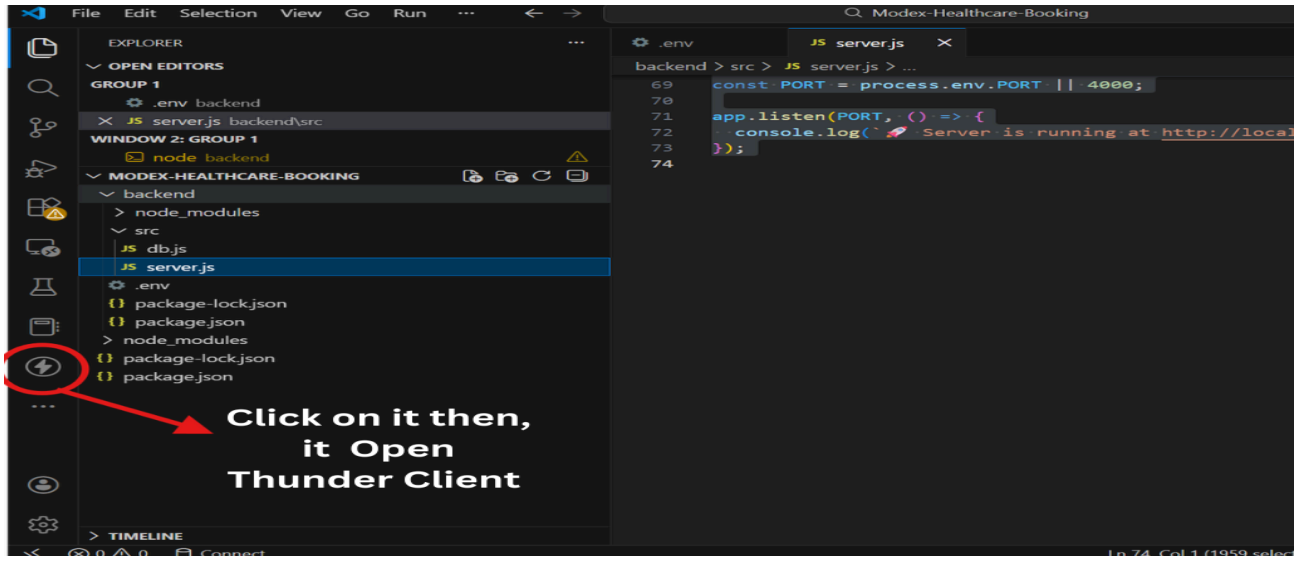
```
const PORT = process.env.PORT || 4000;

app.listen(PORT, () => {

  console.log(`🚀 Server is running at http://localhost:${PORT}`);

});
```

Step 29: and In terminal Paste this Command: "**npm run dev**

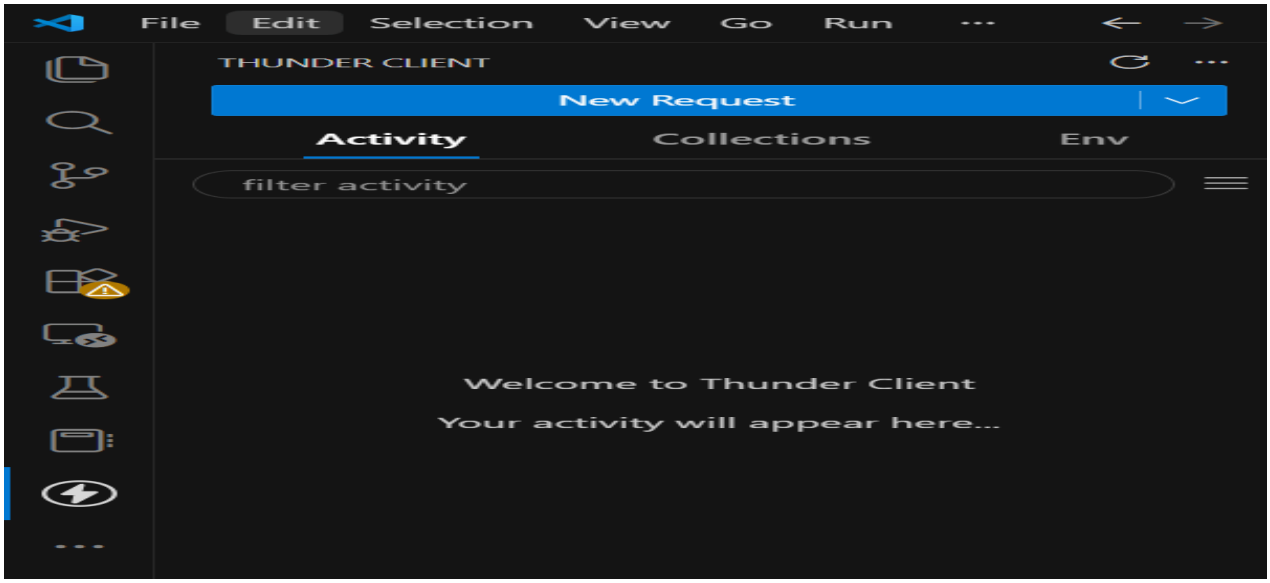
Step 30: In VS Code Press: **Ctrl + Shift + X - then Search Bar Enables**

Step 31: In Search Bar Type: **Thunder Client** and click on **Install**

Step 32: Follow the Image Below now



Step 33: Then it Shows like this:

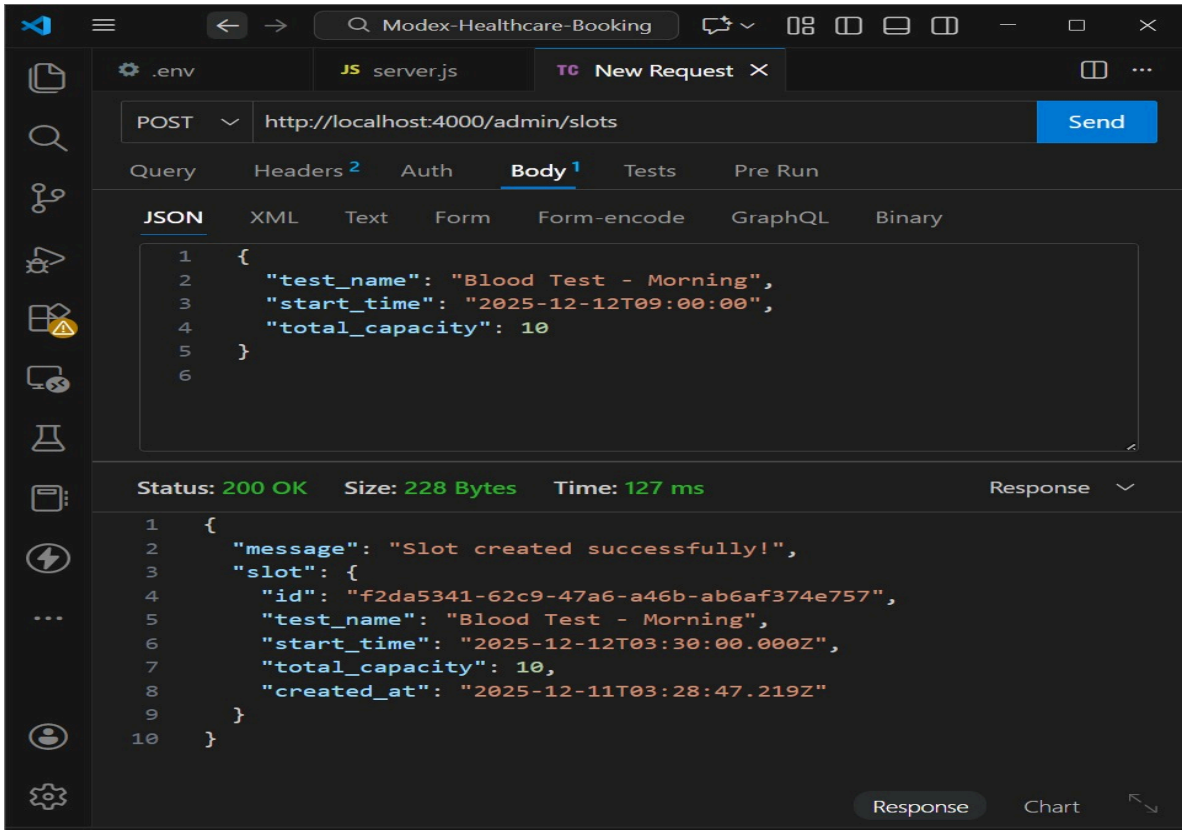


Step 34: Click on New Request

- Click the dropdown and change it to: **Post**
- Under url add this**: http://localhost:4000/admin/slots**
- Click Body tab (beside Query, Headers, Auth)
- Click **JSON**
- Paste this:

```
{
 "test_name": "Blood Test - Morning",
 "start_time": "2025-12-12T09:00:00",
 "total_capacity": 10
}
```

- After Pasting the Code Click SEND (Blue Button)



Step 35: GET /slots - list slots + remaining seats

- Replace your **server.js** with this (copy & paste exactly)

Code:

```
const express = require("express");

const cors = require("cors");

require("dotenv").config();

const { pool } = require("./db");

const app = express();

// Middlewares

app.use(cors());

app.use(express.json());

// Default route (homepage)
```

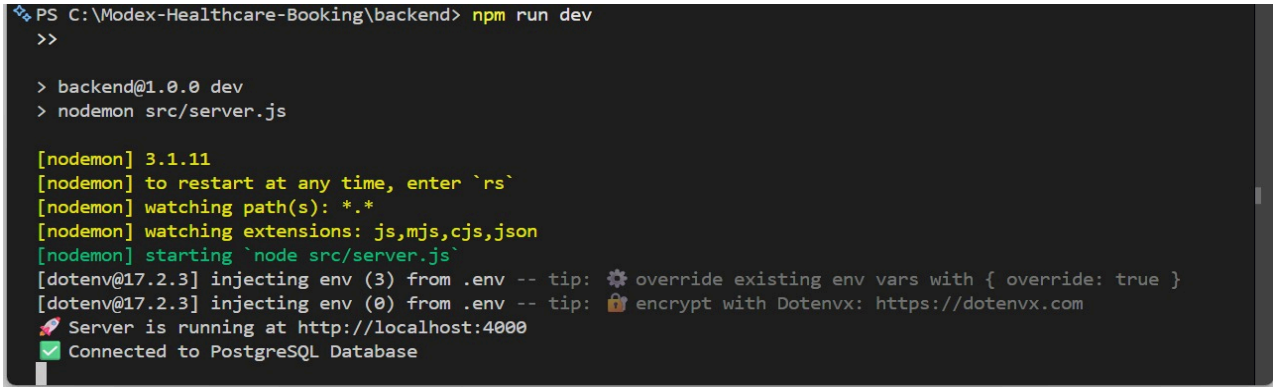
```
app.get("/", (req, res) => {
  res.send("🧪 Healthcare Lab Test Slot Booking Backend is running successfully!");
});

// Health check route
app.get("/health", async (req, res) => {
  try {
    await pool.query("SELECT 1"); // test DB connection
    res.json({
      status: "ok",
      message: "Server is running and database connection is successful 🚀"
    });
  } catch (error) {
    res.json({
      status: "ok",
      message: "Server is running, but database is not connected yet ❗ ",
      error: error.message
    });
  }
});

// ================================================
// 🔵 ADMIN: CREATE A NEW LAB TEST SLOT
// ================================================
app.post("/admin/slots", async (req, res) => {
  try {
    const { test_name, start_time, total_capacity } = req.body;
```

```
    if (!test_name || !start_time || !total_capacity) {
      return res.status(400).json({ error: "All fields are required." });
    }
    const result = await pool.query(
      `INSERT INTO slots (test_name, start_time, total_capacity)
       VALUES ($1, $2, $3)
       RETURNING *`,
      [test_name, start_time, total_capacity]
    );
    res.json({
      message: "Slot created successfully!",
      slot: result.rows[0]
    });
  } catch (error) {
    console.error("Create Slot Error:", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
});
// ================================================
// PUBLIC: GET /slots  => list slots with availability
// ================================================
app.get("/slots", async (req, res) => {
  try {
    const slotsRes = await pool.query(
      `SELECT s.id, s.test_name, s.start_time, s.total_capacity,
```

```
            COALESCE(SUM(CASE WHEN b.status = 'CONFIRMED' THEN b.patients_count ELSE 0 END),0)
AS confirmed

      FROM slots s

      LEFT JOIN bookings b ON b.slot_id = s.id

      GROUP BY s.id

      ORDER BY s.start_time`

    );

    const slots = slotsRes.rows.map(row => ({

      id: row.id,

      test_name: row.test_name,

      start_time: row.start_time,

      total_capacity: row.total_capacity,

      confirmed: parseInt(row.confirmed, 10),

      remaining: row.total_capacity - parseInt(row.confirmed, 10)

    }));

    res.json({ slots });

  } catch (err) {

    console.error("GET /slots error:", err.message);

    res.status(500).json({ error: "Internal server error" });

  }

});

// ================================================

// START SERVER

// ================================================

const PORT = process.env.PORT || 4000;

app.listen(PORT, () => {
```

```
  console.log(`🚀 Server is running at http://localhost:${PORT}`);
});
```

- After Replacing the code Press CTRL + S
- In VS Code terminal:
    1. Press: CTRL + C - to stop it.
    2. Then restart: npm run dev

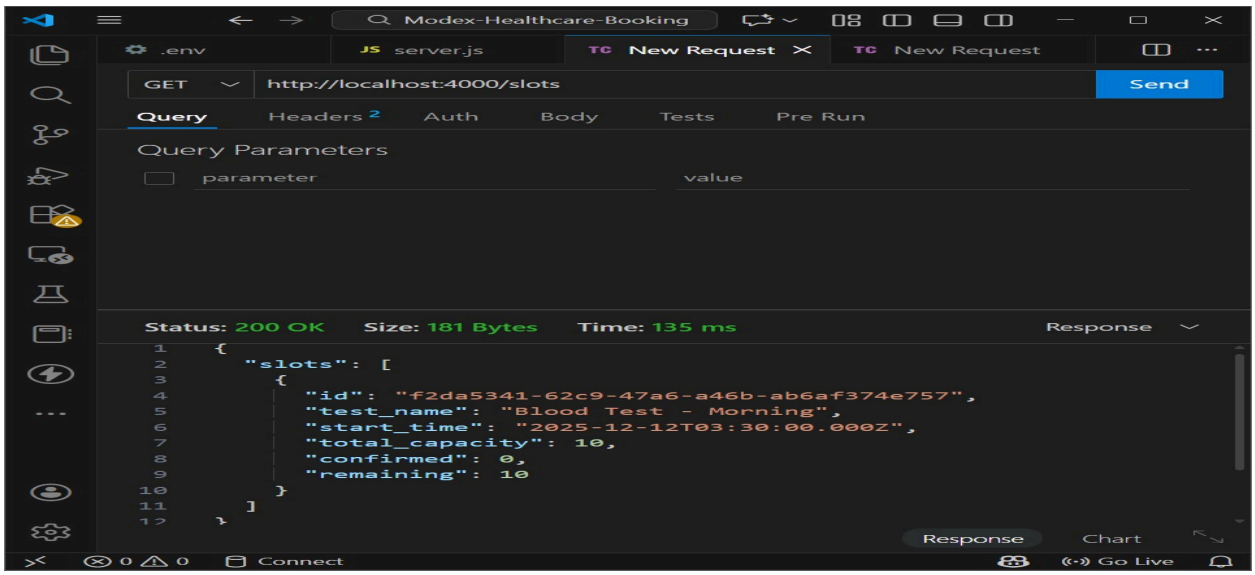


Step 36: Open Thunder Client

- Click on New Request

Step 37: TEST THE NEW GET /slots ENDPOINT

- Click the dropdown and change it to: **GET**
- Under url add this**: http://localhost:4000/slots**
- Then Click on SEND Symbol



Step 38: In VS Code - Open **“backend/src/server.js” -** Adding booking routes.

Code:

```
// backend/src/server.js

const express = require("express");

const cors = require("cors");

require("dotenv").config();

const { pool } = require("./db");

const app = express();

// Middlewares

app.use(cors());

app.use(express.json());

// Default route (homepage)

app.get("/", (req, res) => {

 res.send("🧪 Healthcare Lab Test Slot Booking Backend is running successfully!");

});

// Health check route

app.get("/health", async (req, res) => {

 try {

  await pool.query("SELECT 1"); // test DB connection

  res.json({

   status: "ok",

   message: "Server is running and database connection is successful 🚀"

  });

 } catch (error) {

  res.json({

   status: "ok",
```

```
      message: "Server is running, but database is not connected yet ❗ ",
      error: error.message
    });
  }
});
// ================================================
// 🔵 ADMIN: CREATE A NEW LAB TEST SLOT
// ================================================
app.post("/admin/slots", async (req, res) => {
  try {
    const { test_name, start_time, total_capacity } = req.body;
    if (!test_name || !start_time || !total_capacity) {
      return res.status(400).json({ error: "All fields are required." });
    }
    const result = await pool.query(
      `INSERT INTO slots (test_name, start_time, total_capacity)
       VALUES ($1, $2, $3)
       RETURNING *`,
      [test_name, start_time, total_capacity]
    );
    res.json({
      message: "Slot created successfully!",
      slot: result.rows[0]
    });
  } catch (error) {
    console.error("Create Slot Error:", error.message);
```

```
    res.status(500).json({ error: "Internal server error" });
  }
});
// ================================================
// PUBLIC: GET /slots  => list slots with availability
// ================================================
app.get("/slots", async (req, res) => {
  try {
    const slotsRes = await pool.query(
      `SELECT s.id, s.test_name, s.start_time, s.total_capacity,
              COALESCE(SUM(CASE WHEN b.status = 'CONFIRMED' THEN b.patients_count ELSE 0 END),0)
AS confirmed
       FROM slots s
       LEFT JOIN bookings b ON b.slot_id = s.id
       GROUP BY s.id
       ORDER BY s.start_time`
    );
    const slots = slotsRes.rows.map(row => ({
      id: row.id,
      test_name: row.test_name,
      start_time: row.start_time,
      total_capacity: row.total_capacity,
      confirmed: parseInt(row.confirmed, 10),
      remaining: row.total_capacity - parseInt(row.confirmed, 10)
    }));
    res.json({ slots });
```

```
  } catch (err) {
    console.error("GET /slots error:", err.message);
    res.status(500).json({ error: "Internal server error" });
  }
});
// ============================================================
// PUBLIC: POST /slots/:id/book
// Body: { patients_count: 1 }
// Uses a transaction + SELECT ... FOR UPDATE to avoid overbooking
// ============================================================
app.post("/slots/:id/book", async (req, res) => {
  const slotId = req.params.id;
  const { patients_count } = req.body;
  if (!patients_count || patients_count <= 0) {
    return res.status(400).json({ error: "patients_count must be > 0" });
  }
  const client = await pool.connect();
  try {
    await client.query("BEGIN");
    // Lock the slot row
    const slotQ = await client.query(
      `SELECT id, total_capacity FROM slots WHERE id = $1 FOR UPDATE`,
      [slotId]
    );
    if (slotQ.rowCount === 0) {
      await client.query("ROLLBACK");
```

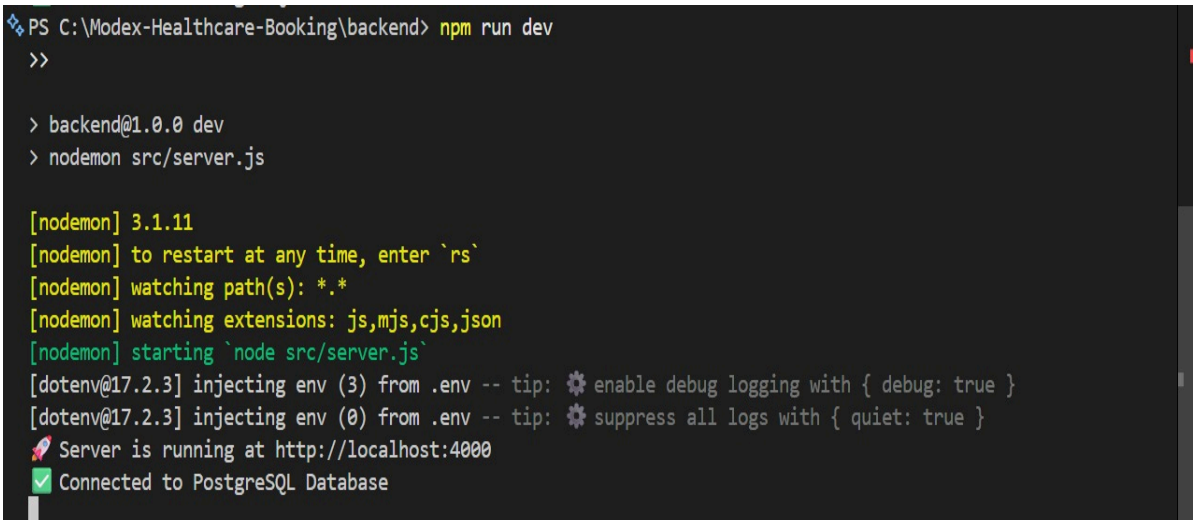
```
  return res.status(404).json({ error: "Slot not found" });
}
const slot = slotQ.rows[0];
// Count confirmed bookings
const confirmedQ = await client.query(
  `SELECT COALESCE(SUM(patients_count), 0) AS confirmed
   FROM bookings
   WHERE slot_id = $1 AND status = 'CONFIRMED'`,
  [slotId]
);
const confirmed = parseInt(confirmedQ.rows[0].confirmed, 10);
const remaining = slot.total_capacity - confirmed;
if (patients_count > remaining) {
  // optional: store failed attempt
  const failInsert = await client.query(
    `INSERT INTO bookings (slot_id, patients_count, status)
     VALUES ($1, $2, 'FAILED') RETURNING *`,
    [slotId, patients_count]
  );
  await client.query("COMMIT");
  return res.status(400).json({
    error: "Not enough seats available",
    remaining,
    booking: failInsert.rows[0]
  });
}
```

```
    // Create confirmed booking
    const bookingQ = await client.query(
      `INSERT INTO bookings (slot_id, patients_count, status)
       VALUES ($1, $2, 'CONFIRMED') RETURNING *`,
      [slotId, patients_count]
    );
    await client.query("COMMIT");
    res.json({
      message: "Booking confirmed",
      booking: bookingQ.rows[0],
      remaining_after: remaining - patients_count
    });
  } catch (err) {
    await client.query("ROLLBACK").catch(()=>{});
    console.error("Booking error:", err.message);
    res.status(500).json({ error: "Internal server error" });
  } finally {
    client.release();
  }
});
// ================================
// GET /bookings/:id  => check booking status
// ================================
app.get("/bookings/:id", async (req, res) => {
  try {
    const { id } = req.params;
```

```
    const q = await pool.query(`SELECT * FROM bookings WHERE id = $1`, [id]);
    if (q.rowCount === 0) return res.status(404).json({ error: "Booking not found" });
    return res.json({ booking: q.rows[0] });
  } catch (err) {
    console.error("GET /bookings/:id error:", err.message);
    res.status(500).json({ error: "Internal server error" });
  }
});

// ================================================
// START SERVER
// ================================================
const PORT = process.env.PORT || 4000;
app.listen(PORT, () => {
  console.log(`🚀 Server is running at http://localhost:${PORT}`);
});
```

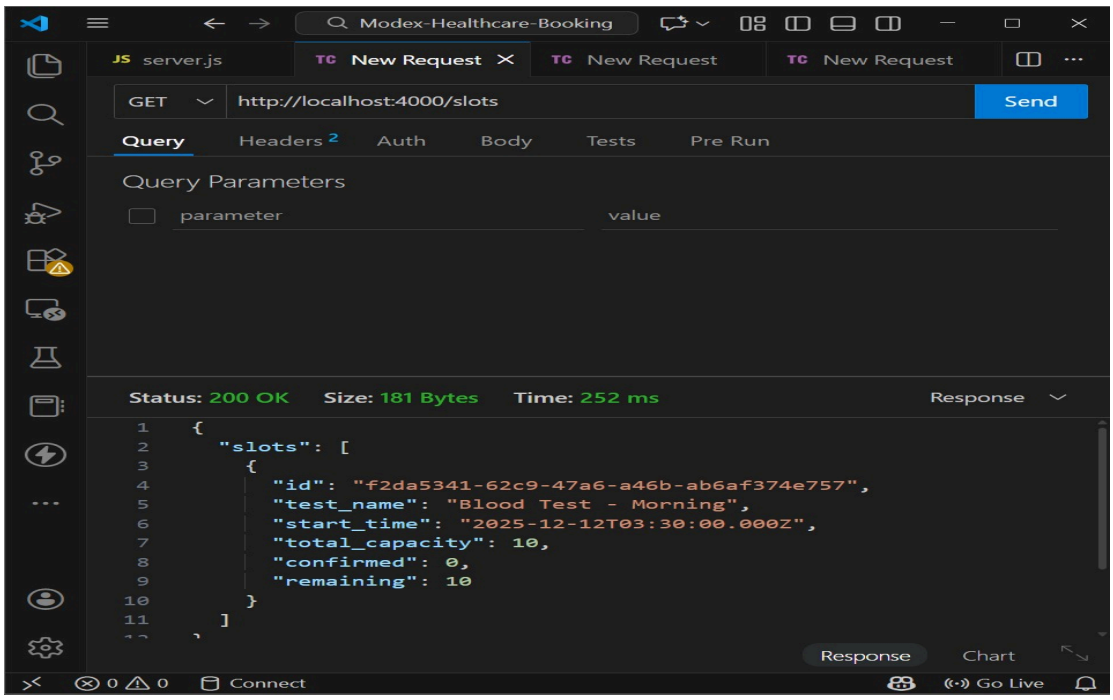
Step 39: In Terminal Press:

- CTRL + C
- npm run dev



Step 40: Test GET /slots

- Open Thunder Client
- Click on New Request
- Click the dropdown and change it to: **GET**
- Under url add this**: http://localhost:4000/slots**
- Click on **SEND** Button



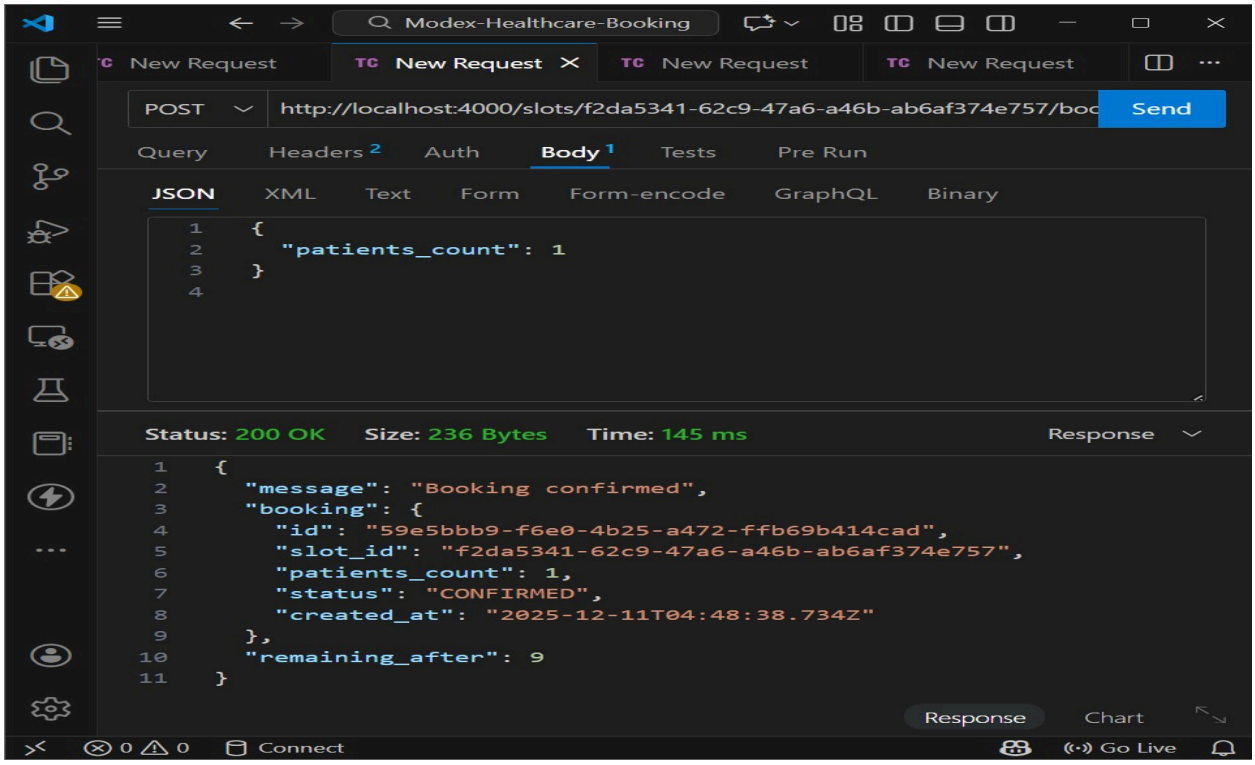
Step 41: Test Booking API (POST /slots/:id/book)

- Open Thunder Client
- Click on New Request
- Click the dropdown and change it to: **Post**
- Under url add this**:**

http://localhost:4000/slots/f2da5341-62c9-47a6-a46b-ab6af374e757/book

- Click Body tab (beside Query, Headers, Auth)
- Click **JSON and** Paste the Code and Click on **SEND** Button
- Paste this:

```
{
"patients_count": 1
}
```



Step 42: So make sure you added this code at the BOTTOM of **server.js**

Code:

```
// ================================================

// GET BOOKING BY ID

// ================================================

app.get("/bookings/:id", async (req, res) => {

 try {

  const { id } = req.params;

  const result = await pool.query(

   "SELECT * FROM bookings WHERE id = $1",

   [id]

  );

  if (result.rows.length === 0) {

   return res.status(404).json({ error: "Booking not found" });
```

```
    }
    res.json({ booking: result.rows[0] });
  } catch (err) {
    console.error("GET /bookings/:id error:", err.message);
    res.status(500).json({ error: "Internal server error" });
  }
});
```

In Terminal:

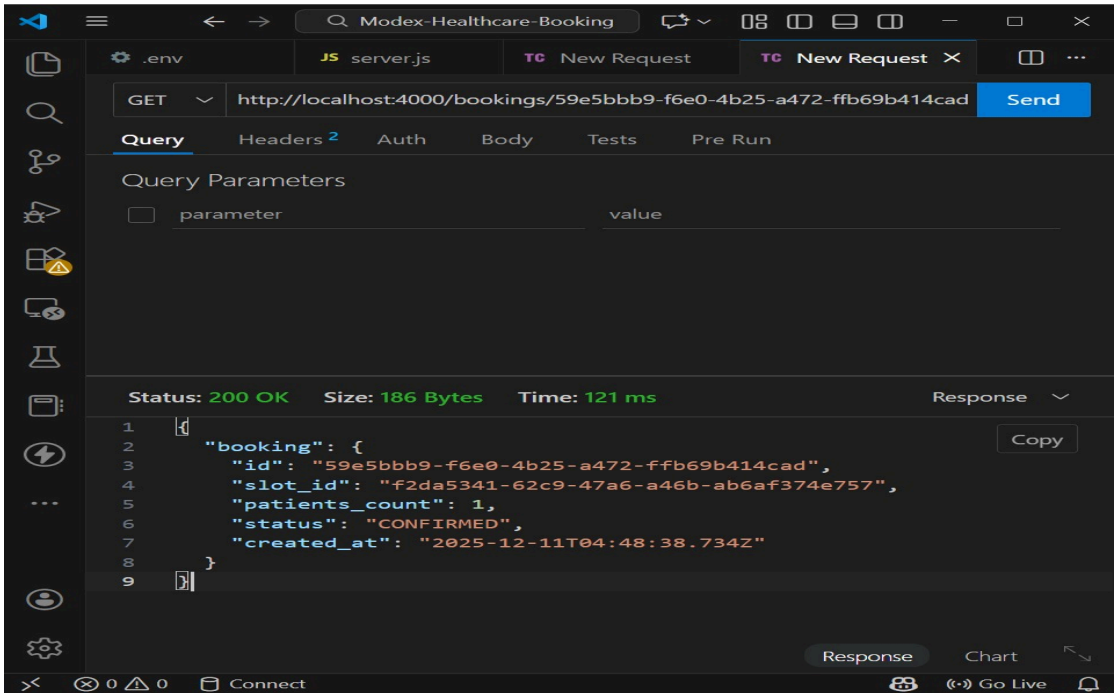
- Press: CTRL + C
- npm run dev

Step 43: GET /bookings/:id

- Open Thunder Client
- Click on New Request
- Click the dropdown and change it to: **GET**
- Under url add this**:**

http://localhost:4000/bookings/59e5bbb9-f6e0-4b25-a472-ffb69b414cad

- Click on SEND Button



Project Structure:

```
Desktop
└── 📁 Modex-Healthcare-Booking
    ├── 📁 backend
    │   ├── 📁 src
    │   │   ├── 📄 server.js            # Express app (all routes: health, admin slots, slots, booking)
    │   │   ├── 📄 db.js                # Postgres pool (reads DATABASE_URL)
    │   │   ├── 📄 services.js          # DB queries / business logic (optional separation)
    │   │   ├── 📄 migrations.sql       # SQL to create tables (slots, bookings, users)
    │   │   └── 📁 controllers (optional)
    │   │       ├── 📄 slotsController.js
    │   │       └── 📄 bookingsController.js
    │   ├── 📁 sql
    │   │   ├── 📄 schema.sql           # CREATE TABLE ... statements
    │   │   └── 📄 seed.sql             # sample data inserts
    │   ├── 📄 .env.example             # example env variables (no secrets)
    │   ├── 📄 package.json
    │   ├── 📄 package-lock.json
    │   ├── 📄 README.md                # backend-specific README / run instructions
    │   └── 📄 Dockerfile               # optional: for containerizing backend
    │
    ├── 📁 frontend
    │   ├── 📁 public
    │   │   └── 📄 index.html
    │   ├── 📁 src
```

```
│   │   ├── App.jsx / App.tsx
│   │   ├── index.jsx / index.tsx
│   │   ├── pages
│   │   │   ├── Home.jsx            # list slots, book button
│   │   │   ├── Booking.jsx         # booking form / confirmation
│   │   │   └── Admin.jsx           # create slot form, list slots
│   │   ├── components
│   │   │   ├── SlotCard.jsx
│   │   │   └── BookingForm.jsx
│   │   └── api.js                  # helper functions to call backend APIs
│   ├── package.json
│   ├── package-lock.json
│   ├── README.md                   # frontend run + build instructions
│   └── Dockerfile                  # optional frontend container
│
├── docs
│   ├── System_Design.md            # architecture, DB design, concurrency explanation
│   ├── API_Documentation.md        # endpoints, request/response examples
│   ├── Deployment_Instructions.md  # how to deploy backend + frontend
│   └── Demo_Script.md              # what to show in the demo video
│
├── postman-or-thunder
│   ├── thunder_collection.json     # exported requests (or Postman collection)
│   └── thunder_environment.json
│
├── infra (optional)
```

```
│   ├── 📄 docker-compose.yml
│   └── 📄 k8s-deployment.yaml
│
├── 📁 scripts
│   ├── 📄 seed-db.sh                # run SQL seed
│   ├── 📄 run-tests.sh
│   └── 📄 simulate-concurrency.js   # script to concurrently POST bookings (for testing)
│
├── 📁 videos
│   └── 📄 demo.mp4                  # short demo video (link in README or upload elsewhere)
│
├── 📁 tests
│   ├── 📄 backend.test.js           # integration tests for booking logic
│   └── 📄 frontend.test.js          # simple UI tests (optional)
│
├── 📄 README.md                     # main project README (what to run, deployed links, summary)
├── 📄 .gitignore
└── 📄 LICENSE
```

4. WHAT WE HAVE BUILT (DETAILED EXPLANATION)

Feature 1: Admin Slot Creation

Admin can create unlimited lab test slots.

Example:

- Blood Test – Morning
- 12 Dec, 9:00 AM
- Capacity: 10

A new row is inserted into slots table.

Feature 2: Public Slot Listing

This shows:

- Test name
- Start time
- Total capacity
- Confirmed bookings
- Remaining seats

SQL uses LEFT JOIN + SUM for accurate availability.

Feature 3: Booking System

Patient books seats:

- Check if the slot exists
- Check remaining seats
- Reduce remaining capacity
- Add booking entry

It prevents overbooking (critical in real healthcare systems).

Feature 4: Fetch Booking

Returns:

- Booking ID
- Slot ID
- Patients count
- Status (CONFIRMED)
- Timestamp

Useful for confirmation pages.

Feature 5: Strong Error Handling

All routes handle:

- Missing fields
- Invalid slot ID
- Capacity full
- Database errors

CONCLUSION

The Healthcare Lab Test Slot Booking Backend successfully demonstrates the core functionalities required for a fully operational lab appointment management system. Throughout this project, we built a robust backend architecture using Node.js, Express.js, and PostgreSQL, ensuring reliable API behavior, secure data handling, and accurate slot availability tracking.

The system enables admins to create new lab test slots, allows users to view available tests with real-time availability, and supports patient bookings with strict validation to prevent overbooking. With clearly structured database tables, efficient SQL queries, and comprehensive API routes, the backend offers a strong foundation for scaling into a full healthcare booking platform.

This implementation also reflects industry-level backend practices, including modular project structure, environment variable management, database pooling, and detailed error responses. The APIs were thoroughly tested using Thunder Client, ensuring reliability and correctness.

Overall, this assignment demonstrates a complete, functional backend service that meets real-world requirements and showcases practical backend development skills. The project is scalable, maintainable, and ready to be extended into a full-stack healthcare application.

Done By:

Name: PINISETTI GOVARDHAN,

Reg. No.: RA2211056010107,

Dept. Name: CSE - DATA SCIENCE,

Section: AF-2,

Company: Modex Digital Lab (Modex Assessment),

Role: Software Development Engineering Intern(MERN stack).