ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

Software Engineering and Development

M1: DIA, OCC, CCC
2025-2026

Lab session 5: Learn JavaScript and Create Your Own REST API

Learning Goals:

- Strengthen your understanding of JavaScript fundamentals used in NodeJS.
- Implement the API routes of your project.

Context: In the previous Lab session, you worked on:

- GitHub Actions (CI/CD pipeline)
- Jira task management
- A NodeJS test backend with some routes

Prerequisites:

- NodeJS installed.
- A working copy of your previous lab (Full DevOps Lab project).
- A functional GitHub repository and Jira board.

You will continue working inside your previous project (Full_DevOps_Lab), using the same CI/CD setup, repository, and Jira board.

1 JavaScript Basics

1.1 Variables

In NodeJS, you'll often use `const` (for values that don't change) and `let` (for modifiable data).

```
const doctorName = "Dr. Anna";
let numberOfPatients = 5;

numberOfPatients += 1; // now 6
console.log(doctorName, numberOfPatients);
```

1.2 Arrays and Objects

APIs usually return JSON, which is based on JavaScript objects and arrays.

```
// Object
const doctor = {
  id: 1,
  name: "Dr. Sarah Lee",
  specialty: "Cardiology"
};

// Array of objects
const doctors = [
  doctor,
  { id: 2, name: "Dr. Amir Khan", specialty: "Pediatrics" }
];

// Access elements
console.log(doctors[0].name); // "Dr. Sarah Lee"
```

1.3 Functions

Functions define reusable logic, used everywhere in Express routes.

```
function greetDoctor(name) {
  return "Hello, ${name}!";
}

console.log(greetDoctor("Dr. Lee"));
```

You can also use **arrow functions**, a common NodeJS style:

```
const greetDoctor = (name) => "Hello, ${name}!";
```

1.4 Modules and Imports

Each route or utility file is a **module**.
You import/export them like this:

```
// utils/hello.js
export const sayHello = (name) => "Hello ${name}";

// routes/hello.route.js
import express from "express";
import { sayHello } from "../utils/hello.js";

const router = express.Router();
router.get("/", (req, res) => res.send(sayHello("Dr. Lee")));
export default router;
```

1.5 Asynchronous Code

API requests can take time. NodeJS uses async/await to handle that smoothly.

```
router.get("/", async (req, res) => {
  const data = await getDoctorsFromDB(); // simulate a database call
  res.json(data);
});
```

2 JavaScript Practice

You'll now train your JavaScript skills before coding your routes.
Each task is independent, but completing all of them will make route development much easier.
Create a file named `js-practice.js` at the root of your project and complete the following exercises.
You can adapt the tasks to your project so that you can use the code later.

2.1 Variables and Data Types

1. Declare three variables using `const` and `let`:
   - a hospital name,
   - a number of doctors,
   - and a Boolean indicating whether the hospital is open.
2. Use `console.log()` to display a message such as: `"Welcome to Central Hospital! We have 12 doctors currently available."`
3. Change the Boolean and log a different message when the hospital is closed.

2.2 Working with Arrays and Loops

1. Create an array `doctors` containing at least five doctor names.
2. Use a `for...of` loop to print each doctor's name with their index:

   ```
   Doctor #1: Dr. Lee
   Doctor #2: Dr. Khan
   ```

3. Add a new doctor using `push()` and print the updated count.
4. Write a function `findDoctor(name)` that returns "Doctor found" or "Doctor not found".

2.3 Objects and Nested Data

1. Create an object `patient`:

```
const patient = {
  name: "Alice Martin",
  age: 34,
  conditions: ["diabetes", "hypertension"],
  doctor: { name: "Dr. Lee", specialty: "Cardiology" }
};
```

2. Display:
   - the patient's doctor name,
   - the number of conditions,
   - and a message: "Alice Martin is treated by Dr. Lee (Cardiology)."
3. Add a new condition dynamically ("anxiety") and display the updated object.

2.4 Functions and Array Filtering

1. Create an array of patient objects:

```
const patients = [
  { id: 1, name: "Alice", age: 34 },
  { id: 2, name: "John", age: 45 },
  { id: 3, name: "Marie", age: 29 }
];
```

2. Write a function `filterByAge(minAge)` that returns all patients older than `minAge`.
3. Write another function `addPatient(name, age)` that adds a new object to the array.
4. Display the updated list.

2.5 Modularization

Split your code into two files:

- `hospitalData.js` → contains arrays and functions.
- `main.js` → imports them and calls the functions.

hospitalData.js

```
export const doctors = ["Dr. Lee", "Dr. Khan"];
export function addDoctor(name) {
  doctors.push(name);
}
```

main.js

```
import { doctors, addDoctor } from "./hospitalData.js";

addDoctor("Dr. Lee");
console.log("Updated doctors:", doctors);
```

Run:

```
node main.js
```

2.6 Error Handling and Validation

Write a function `createAppointment(doctor, patient, date)` that:

- Throws an error if any field is missing.
- Returns an object `doctor, patient, date` otherwise.

```
function createAppointment(doctor, patient, date) {
  if (!doctor || !patient || !date) {
    throw new Error("Missing required fields");
  }
  return { doctor, patient, date };
}

try {
  console.log(createAppointment("Dr. Lee", "Alice", "2025-03-15"));
  console.log(createAppointment("Dr. Lee")); // should trigger error
} catch (err) {
  console.error("Error:", err.message);
}
```

2.7 Asynchronous Code

Simulate fetching hospital data from a database:

```
function getHospitalData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({ name: "Central Hospital", uptime: 1234 });
    }, 1000);
  });
}

async function showData() {
  console.log("Fetching hospital data...");
```

```
  const data = await getHospitalData();
  console.log("Data loaded:", data);
}

showData();
```

- A **Promise** in JavaScript is an object that represents the eventual result of an asynchronous operation (fetching data from the database in this case).
- The `resolve()` call fulfills the promise.
- After 1 second (1000 ms), the promise is resolved with the hospital data.

3 Create Your First APIs (Healthcare Example)

You'll now use what you learned about JavaScript to extend your Full_DevOps_Lab project.

Each team must keep their existing Jira board and add routes and logic matching their Epics (if missing).

Each student should implement and test one route.

Make your repository Public and add its URL to the Excel file: SED Groups.

Example Routes to Implement:

Feature	Description	Example Path
Doctor Management	View and add doctors	`/api/doctors`
Patient Management	View and add patients	`/api/patients`
Appointments	Schedule visits between doctor and patient	`/api/appointments`
Metrics	Return system and hospital stats	`/api/metrics`

3.1 Example: `doctors.route.js`

```
import express from "express";
const router = express.Router();

const doctors = [
  { id: 1, name: "Dr. Sarah Lee", specialty: "Cardiology" },
  { id: 2, name: "Dr. Amir Khan", specialty: "Pediatrics" }
];

// GET all doctors
router.get("/", (req, res) => res.status(200).json(doctors));

// POST a new doctor
```

```
router.post("/", (req, res) => {
  const { name, specialty } = req.body;
  if (!name || !specialty) {
    return res.status(400).json({ error: "Missing required fields"
       });
  }
  const newDoctor = { id: doctors.length + 1, name, specialty };
  doctors.push(newDoctor);
  res.status(201).json(newDoctor);
});

export default router;
```

3.2 Example: patients.route.js

```
import express from "express";
const router = express.Router();

let patients = [
  { id: 1, name: "Alice", age: 30 },
  { id: 2, name: "John", age: 45 }
];

// GET all patients
router.get("/", (req, res) => res.json(patients));

// POST new patient
router.post("/", (req, res) => {
  const { name, age } = req.body;
  if (!name || !age)
    return res.status(400).json({ error: "Missing fields" });
  const newPatient = { id: patients.length + 1, name, age };
  patients.push(newPatient);
  res.status(201).json(newPatient);
});

export default router;
```

3.3 Testing Example: test/doctors.test.js

```
import request from "supertest";
import app from "../src/app.js";

describe("Doctors API", () => {
  it("GET /api/doctors should return an array", async () => {
```

```
    const res = await request(app).get("/api/doctors");
    expect(res.status).toBe(200);
    expect(Array.isArray(res.body)).toBe(true);
  });

  it("POST /api/doctors should add a new doctor", async () => {
    const res = await request(app)
      .post("/api/doctors")
      .send({ name: "Dr. Lee", specialty: "Neurology" });
    expect(res.status).toBe(201);
    expect(res.body).toHaveProperty("id");
  });
});
```

In this code:

- Each `it(...)` block defines a single test case.
- `request(app).get("/api/doctors")` : simulates sending a GET request to our Express app.
- `await` : waits for the response asynchronously.
- `res`: contains the full HTTP response object:

```
{
  status: 200,
  body: [ { id: 1, name: "Dr. Lee", specialty: "Cardiology"
      } ],
  headers: {...}
}
```

- `expect(res.status).toBe(200)` : checks if the response code is 200.
- `expect(Array.isArray(res.body)).toBe(true)` : checks if the response body is indeed an array.
- If both expectations pass, the test succeeds. If any fails, the test fails.
- `.post()` sends a POST request to your `/api/doctors` endpoint.
- `.send(...)` sends a JSON body.
- The new doctor should have an auto-generated id, checked with `.toHaveProperty("id")`.

4 Test and Validate Your API with Postman

Objectives:

- Learn to send requests and inspect responses using Postman.
- Verify your API routes return the expected data and status codes.

4.1 Install Postman

- Go to the Postman download page.
- Install and open the desktop app.

4.2 Create a New Collection

Create a collection named **Healthcare API Tests** (for example).
Add four requests corresponding to your routes:

Method	Endpoint	Description
GET	`/api/doctors`	Retrieve all doctors
POST	`/api/doctors`	Add a new doctor
GET	`/api/patients`	Retrieve all patients
POST	`/api/appointments`	Add a new appointment

Set the base URL:

```
http://localhost:3000
```

4.3 Test GET Requests

1. Run your project locally with:

   ```
   npm run dev
   ```

2. Open the GET /api/doctors request.
3. Click Send.
4. You should receive:

   ```
   [
     { "id": 1, "name": "Dr. Sarah Lee", "specialty": "
        Cardiology" },
     { "id": 2, "name": "Dr. Amir Khan", "specialty": "
        Pediatrics" }
   ]
   ```

5. Observe the Status (should be 200 OK) and Response Time.

4.4 Test POST Requests

Switch to **Body** → **raw** → **JSON** and enter:

```
{
  "name": "Dr. Julie",
  "specialty": "Neurology"
}
```

Click **Send**. You should get a `201 Created` response:

```
{
  "id": 3,
  "name": "Dr. Julie",
  "specialty": "Neurology"
}
```

Repeat for `/api/patients` or `/api/appointments` as needed.

4.5 Automate Tests in Postman

Go to the **Tests** tab in Postman for each request, and add assertions:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response is an array", function () {
    pm.expect(pm.response.json()).to.be.an("array");
});
```

Run the entire collection. All tests should pass.

4.6 Export and Submit

Once tests pass:

1. Click on your collection → "Export".
2. Choose version 2.1 (recommended).
3. Name it Healthcare_API_Postman_Collection.json.
4. Upload it to your team's GitHub repo under /postman/.

5 DevOps Workflow Recapitulation

5.1 Branch

```
git checkout -b feature/DEVOPS-15-patient-route
```

5.2 Commit

```
git commit -m "feat(DEVOPS-15):  add /api/patients route"
```

5.3 Push

```
git push origin feature/DEVOPS-15-patient-route
```

5.4 Pull request

- Linked to your Jira story
- Merge after CI passes

6 Continuous Integration

Your `.github/workflows/ci.yml` already runs:

```
 - run:  npm ci
- run:  npm run lint
- run:  npm test -- --coverage
```

Keep coverage $\geq 80\%$.
All new routes must be tested.