Web Application Architecture

Lab 2 – Create an API

# 

# **Express Lab Setup**

## Install Node and NPM

Download Node.js from <https://nodejs.org/en/download/>.

This will contain both Node and NPM. Select the Long Term Support (LTS) option:

* Node.js v22.x
* npm v11.x

(Note: these were the current LTS versions at the time of updating this document – 22nd Sept 2024)

Open a command prompt / terminal window and check the version of node js by entering the commands:

* node -version
* npm -version

## Code editor (Visual Studio Code)

For this lab work we will be using Visual Studio Code as the code editor for our web projects.

Download and install Visual Studio Code:

* <https://code.visualstudio.com/download>

# **Application Requirements**

We have been tasked with creating an API for student data.

The API should offer CRUD operations for student data.

A Student consists of:

* First name
* Last name
* Student Number
* Email
* Course

The API should follow best practices where possible.

# **Initial Setup**

Step 0: Create project folder

Create a folder named students\_crud\_api

Open the folder in VS Code

**Step 1: Initialize the project**

Open a terminal in VS Code (From the menu in VS Code: Terminal -> New Terminal)

Run this command in the terminal window: npm init -y

In an Express (or any Node.js) project, running **npm init -y** does the following:

* Initializes a new Node.js project in the current directory.
* Creates a package.json file automatically, without asking you interactive questions (like project name, version, description, entry point, etc.).
* The -y (or --yes) flag tells npm to use the default values for all prompts.

**Step 2: Install Dependencies**

We will implement the API using Express and Typescript.

The following dependencies will be used to offer functionality in the application.

**express**

* The core web framework for Node.js.
* Provides tools for building APIs and web applications (routing, middleware, request/response handling).

**cors**

* Middleware that enables Cross-Origin Resource Sharing (CORS).
* Lets your API be accessed by web apps running on different domains (e.g., frontend http://localhost:3000 calling backend http://localhost:5000).

**sqlite**

* A lightweight SQL database engine written in C, often used for prototyping or small apps.
* This package provides the SQLite database binary itself.

**sqlite3**

* Node.js bindings for SQLite.
* Lets your Express app connect to and query an SQLite database.

**uuid**

* Utility library for generating unique identifiers (UUIDs).
* Useful for creating unique keys for database records, session IDs, etc.

**swagger-jsdoc**

* Parses JSDoc-style comments in your code to automatically generate OpenAPI (Swagger) documentation.
* Makes it easier to keep your API docs in sync with your code.

**swagger-ui-express**

* Middleware for serving a Swagger UI interface in Express.
* Lets you visualize and test your API endpoints in a browser (interactive API docs).
* Usually combined with swagger-jsdoc.

**In terminal run the following command to install the dependencies:**

npm install express cors sqlite sqlite3 uuid swagger-jsdoc swagger-ui-express

**Step 3: Install Development Dependencies**

**Development dependencies (-D means they’re not required in production but only while developing).**

**typescript**

* The TypeScript compiler (tsc).
* Translates TypeScript (.ts) into JavaScript (.js) so Node.js can run it.
* Also provides type-checking for safer code.

**ts-node-dev**

* Like nodemon but for TypeScript.
* Runs .ts files directly without manual compilation.
* Automatically restarts the server when you change your code.

**@types/express**

* Type definitions for Express.
* Lets TypeScript know the shape of Express objects (e.g., Request, Response) so you get IntelliSense and type safety.

**@types/cors**

* Type definitions for the cors middleware.
* Ensures correct typing when using app.use(cors()).

**@types/uuid**

* Type definitions for the uuid library.
* Helps TypeScript understand functions like uuidv4().

**@types/node**

* Type definitions for core Node.js APIs (like fs, path, process, etc.).
* Essential for writing TypeScript in a Node environment.

**In terminal run the following command to install the development dependencies:**

npm install -D typescript ts-node-dev @types/express @types/cors @types/uuid @types/node

npm i --save-dev @types/express

npm i --save-dev @types/cors

**Step 4: Setup the Project Structure**

In VSCode, Create all the files and folders listing below.

|  |  |
| --- | --- |
| student-api/  ├── src/  │ ├── index.ts # Entry point  │ ├── db.ts # SQLite setup  │ ├── routes/  │ │ └── students.ts # CRUD routes  │ └── types/  │ ├── swagger-jsdoc.d.ts  │ └── swagger-ui-express.d.ts  ├── package.json  └── tsconfig.json | A screen shot of a computer  AI-generated content may be incorrect. |

**Step 5: Setup TypeScript**

Add the following to tsconfig.json:

|  |  |
| --- | --- |
| {  "compilerOptions": {  "target": "ES2020",  "module": "commonjs",  "outDir": "dist",  "rootDir": "src",  "strict": true,  "esModuleInterop": true,  "allowSyntheticDefaultImports": true,  "typeRoots": [  "./src/types",  "./node\_modules/@types"  ]  },  "include": [  "src/\*\*/\*.ts"  ],  "exclude": [  "node\_modules",  "dist"  ]  } | A screen shot of a computer program  AI-generated content may be incorrect. |

This is a TypeScript configuration file (tsconfig.json).

This is a description of the compiler options listing in tsconfig.json

**target: "ES2020"**

* Tells TypeScript to compile your code down to ECMAScript 2020 JavaScript.
* Defines what JavaScript features will be available in the output.

**module: "commonjs"**

* Specifies the module system.
* commonjs is the standard for Node.js (so you can use require/module.exports).
* Necessary for running with Node before full ES module support.

**outDir: "dist"**

* Where the compiled JavaScript files will be placed.
* Example: src/index.ts → dist/index.js.

**rootDir: "src"**

* Tells the compiler where your TypeScript source files live.
* Ensures that output mirrors the folder structure of src.

**strict: true**

* Enables strict type checking (best practice).
* Catches potential bugs early by enforcing things like non-null checks and strict function typing.

**esModuleInterop: true**

* Makes it easier to work with CommonJS modules in TypeScript.

**typeRoots: ["./src/types", "./node\_modules/@types"]**

* Tells the compiler where to look for type definitions.
* Useful if you add your own custom .d.ts files in src/types.

Summary: Take the TypeScript files from src/, compile them to modern JavaScript (ES2020) using CommonJS, output to dist/, apply strict type checking, and use both built-in and custom type definitions

**Step 6: Setup Express**

Add the following code to index.ts

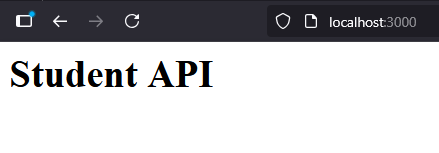
|  |  |
| --- | --- |
| import express from "express";  import bodyParser from "body-parser";  const app = express();  const PORT = 3000;  app.use(bodyParser.json());  // Test to see the server working in the browser  app.get("/", (req,res) => {      res.send("<h1>Student API</h1>")  })  app.listen(PORT, () => {      console.log(`Server running at http://localhost:${PORT}`);  }); |  |

**Run the server, In terminal run:**

npx ts-node-dev src/index.ts

**Test the server:**

Open a browser and go to: localhost:3000



Note: To stop running the server click into the terminal window and press ctrl and c together

**Step 7: Setup the database**

For this example we will use Sqlite to offer data persistence for our student API.

Add the following code to db.ts

|  |  |
| --- | --- |
| import sqlite3 from "sqlite3";  import { open } from "sqlite";  export async function openDb() {  return open({  filename: "./students.db",  driver: sqlite3.Database,  });  }  export async function initDb() {  const db = await openDb();  await db.exec(`  CREATE TABLE IF NOT EXISTS students (  id TEXT PRIMARY KEY,  firstName TEXT NOT NULL,  lastName TEXT NOT NULL,  studentNumber TEXT NOT NULL,  email TEXT NOT NULL,  course TEXT NOT NULL  )  `);  return db;  } |  |

This code sets up a SQLite database for managing student records. It provides two functions:

* openDb(): Opens (or creates if missing) a students.db database file using SQLite.
* initDb(): Ensures the database has a students table, creating it if it doesn’t exist. The table stores each student’s ID, first name, last name, student number, email, and course.

**Step 8: Add the database to index.ts**

In index.ts, add an import to include the db functionality:

import { initDb } from "./db";

Next we need to call initDb to setup the Sqlite database.

Update the app.listen code block as follows:

// Start server

initDb().then(() => {

    app.listen(PORT, () => {

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

    });

});

This code initializes the database and then starts the server. The InitDb function from db.ts is called, it will create the students table if it doesn’t exist. The open connection is returned and can now be used in the application. Once the database setup is complete, the server listens on the specified port and logs the URL where it’s running.

**Step 9: Setup the Student GET route**

Add the following code to students.ts

|  |  |
| --- | --- |
| import { Router } from "express";  import { v4 as uuidv4 } from "uuid";  import { openDb } from "../db";  const router = Router();  export default router; |  |

This code creates an Express router that will handle API routes. It imports:

* Router from Express to define route endpoints,
* uuidv4 to generate unique IDs for new records (e.g., students), and
* openDb from the database module to connect and run queries on the SQLite database.

With the initial setup complete we can now create the students GET API endpoint.

Add the following to students.ts (before export default router)

// GET all students

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

    const db = await openDb();

    const students = await db.all("SELECT \* FROM students");

    res.json({ value: students });

});

This defines a GET endpoint at the root path ("/") of the router.

It first opens a connection to the database using openDb().

Then it runs a query (SELECT \* FROM students) to fetch all student records from the students table.

Finally, it responds with the data as JSON, wrapping the list of students in an object under the key value.

**Step 10: Add routing functionality to index.ts**

**Add the following code to index.ts**

import studentsRouter from "./routes/students";

// Routes

app.use("/v1/students", studentsRouter);

Note: (make sure the app.use is added before InitDb.then)

**Step 11: Test the Student GET route**

Run the server: npx ts-node-dev src/index.ts

Open Postman

A screenshot of a computer

AI-generated content may be incorrect.

We get no students as there are no student entries in the database

**Step 12: Setup the Student POST route**

Add the following code to students.ts (before export default router)

// Create a new student

router.post("/", async (req, res) => {

    const { firstName, lastName, studentNumber, email, course } = req.body;

    const id = uuidv4();

    const db = await openDb();

    await db.run(

        "INSERT INTO students (id, firstName, lastName, studentNumber, email, course) VALUES (?, ?, ?, ?, ?, ?)",

        id, firstName, lastName, studentNumber, email, course

    );

    const newStudent = await db.get("SELECT \* FROM students WHERE id = ?", id);

    res.status(201).json(newStudent);

});

This defines a POST endpoint at the root path ("/") of the students router.

* It extracts student details (firstName, lastName, studentNumber, email, course) from the request body.
* It generates a unique id using uuidv4().
* It connects to the database with openDb() and inserts a new record into the students table.
* After insertion, it retrieves the newly created student by its id.
* Finally, it responds with the new student as JSON, setting the HTTP status to 201 (Created).

**Step 13: Test the Student POST route**

Run the server: npx ts-node-dev src/index.ts

Open Postman

Create a new request

A screenshot of a computer

AI-generated content may be incorrect.

{

"firstName": "Wayne",

"lastName": "Scales",

"studentNumber": "D123456",

"email": "wayne@tudublin.ie",

"course": "TU059"

}

Above is the data added to the Postman request body.

Above we are posting a JSON student object to the server. The server will process the request and access the JSON data via body.req. The server responds with a 201 status and the data that was stored in the db with the new UUID.

**Step 14: Create the Student Update route**

Add the following code to students.ts (before export default router)

// Update a Student Record

router.put("/:id", async (req, res) => {

const { firstName, lastName, studentNumber, email, course } = req.body;

const db = await openDb();

await db.run(

"UPDATE students SET firstName=?, lastName=?, studentNumber=?, email=?, course=? WHERE id=?",

firstName, lastName, studentNumber, email, course, req.params.id

);

const updatedStudent = await db.get("SELECT \* FROM students WHERE id = ?", req.params.id);

res.json(updatedStudent);

});

This defines a PUT endpoint at "/:id" (where :id is the student’s unique identifier).

It reads the updated student details (firstName, lastName, studentNumber, email, course) from the request body.

It connects to the database with openDb() and runs an UPDATE query to modify the matching student record by id.

After updating, it fetches the updated student record from the database.

Finally, it sends the updated student back as a JSON response.

This route lets clients update an existing student’s details by providing the student’s id in the URL and the new data in the request body.

**Step 15: Test the Student Update route**

Run the server: npx ts-node-dev src/index.ts

Open Postman

Create a new request (PUT)

A screenshot of a computer

AI-generated content may be incorrect.

To update a Student we will need to know their Student ID.

The ID is added to the URL as a parameter, we can get this from testing a GET request for list all students.

* Eg: localhost:3000/v1/students/5d2b9d73-5f6d-4915-801e-d9c9e025d015

{

"firstName": "Wayno",

"lastName": "Scales",

"studentNumber": "D123456",

"email": "wayne@tudublin.ie",

"course": "TU059"

}

Above is the data added to the Postman request body.

Above we are posting a JSON student object to the server. The server will process the request and access the JSON data via body.req. The server responds with a 200 status and the data that was updated in the db.

**Step 16: Create the Student Delete route**

Add the following code to students.ts (before export default router)

// Delete a Student

router.delete("/:id", async (req, res) => {

const db = await openDb();

await db.run("DELETE FROM students WHERE id=?", req.params.id);

res.status(204).send();

});

This defines a DELETE endpoint at "/:id", where :id is the student’s unique identifier.

It connects to the database using openDb().

It executes a DELETE query to remove the student record that matches the given id.

After deletion, it responds with HTTP status 204 (No Content), meaning the request succeeded but there’s no response body.

This route allows clients to remove a student from the database by their ID.

**Step 17: Test the Student Delete route**

Run the server: npx ts-node-dev src/index.ts

Open Postman

Create a new request (DELETE)

A screenshot of a computer

AI-generated content may be incorrect.

We provide the ID of the Student as a URL parameter

We get the status code 204, no content

**Step 18: Create the individual Student GET route**

Add the following code to students.ts (before export default router)

// GET Individual Student

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

const db = await openDb();

const student = await db.get("SELECT \* FROM students WHERE id = ?", req.params.id);

if (!student) return res.status(404).json({ message: "Student not found" });

res.json(student);

});

This defines a GET endpoint at "/:id", where :id is the student’s unique identifier.

It connects to the database using openDb().

It runs a SELECT query to find the student with the matching id.

If no student is found, it returns an HTTP 404 (Not Found) response with a JSON error message.

If the student exists, it returns the student’s record as JSON.

This route retrieves a single student by ID and handles the case where the student doesn’t exist.

**Step 19: Test the individual Student GET route**

Run the server: npx ts-node-dev src/index.ts

Open Postman

Create a new request (GET)

We will need to ensure we have created some Students to test this.

A screenshot of a computer

AI-generated content may be incorrect.

The student id is passed in the URL as a parameter.

**Step 20: Setup Swagger**

Add the following to src/types/swagger-jsdoc.d.ts

declare module "swagger-jsdoc" {

import { OpenAPIV3 } from "openapi-types";

export interface Options {

definition: OpenAPIV3.Document;

apis: string[];

}

export default function swaggerJsdoc(options: Options): OpenAPIV3.Document;

}

A screen shot of a computer code

AI-generated content may be incorrect.

**Step 21: Setup Swagger**

Add the following to src/types/swagger-ui-express.d.ts

declare module "swagger-ui-express" {

import { RequestHandler } from "express";

interface SwaggerUIOptions {

explorer?: boolean;

swaggerOptions?: Record<string, any>;

customCss?: string;

customJs?: string;

customfavIcon?: string;

swaggerUrl?: string;

}

export const serve: RequestHandler[];

export function setup(swaggerDoc: any, options?: SwaggerUIOptions): RequestHandler;

}

A screen shot of a computer program

AI-generated content may be incorrect.

**Step 22: Setup Swagger (update index.ts)**

**Add the following imports to index.ts**

import swaggerJsdoc from "swagger-jsdoc";

import swaggerUi from "swagger-ui-express";

**Add the following to index.ts** (after app.use(bodyParser.json()))

// Swagger setup

const swaggerOptions = {

definition: {

openapi: "3.0.0",

info: {

title: "Student API",

version: "1.0.0",

description: "API to manage students",

},

servers: [{ url: `http://localhost:${PORT}` }],

paths: {}, // ✅ prevents TS error

},

apis: ["./src/routes/\*.ts"],

};

const swaggerSpec = swaggerJsdoc(swaggerOptions);

app.use("/docs", swaggerUi.serve, swaggerUi.setup(swaggerSpec));

**Step 23: Setup Swagger (routing)**

Add the following to students.ts (just above export default router)

/\*\*

\* @swagger

\* /v1/students:

\* get:

\* summary: Get all students

\* tags: [Students]

\* responses:

\* 200:

\* description: List of students

\*/

/\*\*

\* @swagger

\* /v1/students/{id}:

\* get:

\* summary: Get a student by ID

\* tags: [Students]

\* parameters:

\* - in: path

\* name: id

\* required: true

\* schema:

\* type: string

\* description: The student ID

\* responses:

\* 200:

\* description: Student found

\* 404:

\* description: Student not found

\*/

/\*\*

\* @swagger

\* /v1/students:

\* post:

\* summary: Create a new student

\* tags: [Students]

\* requestBody:

\* required: true

\* content:

\* application/json:

\* schema:

\* type: object

\* required:

\* - firstName

\* - lastName

\* - studentNumber

\* - email

\* - course

\* properties:

\* firstName:

\* type: string

\* lastName:

\* type: string

\* studentNumber:

\* type: string

\* email:

\* type: string

\* course:

\* type: string

\* responses:

\* 201:

\* description: Student created

\*/

/\*\*

\* @swagger

\* /v1/students/{id}:

\* put:

\* summary: Update a student by ID

\* tags: [Students]

\* parameters:

\* - in: path

\* name: id

\* required: true

\* schema:

\* type: string

\* description: The student ID

\* requestBody:

\* required: true

\* content:

\* application/json:

\* schema:

\* type: object

\* properties:

\* firstName:

\* type: string

\* lastName:

\* type: string

\* studentNumber:

\* type: string

\* email:

\* type: string

\* course:

\* type: string

\* responses:

\* 200:

\* description: Student updated

\* 404:

\* description: Student not found

\*/

/\*\*

\* @swagger

\* /v1/students/{id}:

\* delete:

\* summary: Delete a student by ID

\* tags: [Students]

\* parameters:

\* - in: path

\* name: id

\* required: true

\* schema:

\* type: string

\* description: The student ID

\* responses:

\* 204:

\* description: Student deleted

\* 404:

\* description: Student not found

\*/

**Step 23: Swagger (demo)**

This Swagger documentation describes the Students API, which provides full CRUD operations for managing student records. It includes:

GET /v1/students → Retrieve all students

GET /v1/students/{id} → Retrieve a student by ID

POST /v1/students → Create a new student

PUT /v1/students/{id} → Update an existing student by ID

DELETE /v1/students/{id} → Remove a student by ID

The API lets clients create, read, update, and delete student records in the database.

Run the server

API → http://localhost:3000/v1/students

Swagger Docs → http://localhost:3000/docs

A screenshot of a computer

AI-generated content may be incorrect.