PRACTICAL LESSON PLAN

Subject:	Web Applications Development	**Degree:**	Informatics Engineering
Topic:	REST API: Server	**Professor:**	Noélia Correia (ncorreia@ualg.pt)
Type:	☒ Programming ☐ Setup	**Institution:**	Universiy of Algarve

Goals

In this lab you will **buid a very simple webmail system**. This tutorial, provided in [1], sheds light on the steps involved in developing a REST API that stores data in a DB. As future work you will be asked to develop your own REST API, so take a close look at this example. Overall view of the web application:

- The webmail server side acts as a proxy by communicating with the mail server, so the client should communicate with the webmail server.
- STMP is the communication protocol in use (to send emails), and related SMTP server information is stored in a text file.
- The webmail server allows the client to maintain contacts and send emails.

You have to implement the following server side REST API (see if you really understand it) and test it using **Postman** or **cURL**. You must deliver your code with comments attached to each block of code. A small report with test results must be included too, at the main directory of your project.

Tasks

Part I: Set up your project

Create you Node project and include Typescript as a development dependency. Include:

- Modules as non development dependencies:
  - **express**: build API easily.
  - **nedb**: data storage.
  - **nodemailer**: SMTP functionality.
- Types as development dependencies:
  - Node.js's TypeScript bindings: `@types/node`.
  - TypeScript bindings for the other dependencies: check for them in `https://www.typescriptlang.org/dt/search?search=`.

Part II: Set up a better development environment

- Inside `package.json` file, include a script for source code compilation followed by the start up of the server (with &&, the second command is executed if the first has executed successfully). This allows the use of `npm run compile`, if the server is not running.

```
scripts: { "compile" : "npx tsc && node ./build/main.js" }
```

- Nodemon is a module that can monitor source files for changes, and automatically restart the app. For example, after `npm install --save-dev nodemon`, something like `./node_modules/nodemon/bin/nodemon.js ./build/main.js` could be done. To include it as a script, and use `nodemon -e` option to set the watchable file extensions, the following must be included in the `package.json` file.

```
"scripts": {
  "compile": "npx tsc && node ./build/main.js",
  "dev": "node ./node_modules/nodemon/bin/nodemon.js -e ts --exec \"npm run
compile\""
}
```

The command `npm run dev` can be used now. Note that `main.js` is the entry point of the application.

Part III: `main.ts`*, the entry point*

Entry point (e.g., `index.ts`, `main.ts`; defined when project was created) is the file that is invoked when consumers of your module import it. This includes the main logic for your module, and should include the following:

◇ Import of modules `path` (core Node module) and `express` (together with types).

```
import path from "path";
import express, { Express, NextFunction, Request, Response } from "express";
```

◇ Import of application modules (together with types).

```
import { serverInfo } from "./serverInfo";
import * as SMTP from "./SMTP";
import * as Contacts from "./contacts";
import { IContact } from "./contacts";
```

◇ Create an **Express app** and add some **middleware/functionality** to it.

```
const app: Express = express();
app.use(express.json());
```

◇ Provide **RESTful endpoint** that the client application can call, and serve the code.

```
app.use("/",
  express.static(path.join(__dirname, "../../client/dist"))
);
```

◇ Set CORS security mechanism.

```
app.use(function(inRequest: Request, inResponse: Response,
inNext: NextFunction) {
  inResponse.header("Access-Control-Allow-Origin", "*");
  inResponse.header("Access-Control-Allow-Methods",
    "GET,POST,DELETE,OPTIONS"
  );
  inResponse.header("Access-Control-Allow-Headers",
    "Origin,X-Requested-With,Content-Type,Accept"
  );
  inNext();
});
```

◇ Register path and method for the endpoint that is used to **send messages**.

```
app.post("/messages",
  async (inRequest: Request, inResponse: Response) => {
    try {
      const smtpWorker: SMTP.Worker = new SMTP.Worker(serverInfo);
      await smtpWorker.sendMessage(inRequest.body); // object created by express.json
          middleware
      inResponse.send("ok");
    } catch (inError) {
      inResponse.send("error");
    }
  }
);
```

◇ Register path and method for the endpoint that is used to get **list of contacts** resource.

```
app.get("/contacts",
  async (inRequest: Request, inResponse: Response) => {
    try {
      const contactsWorker: Contacts.Worker = new Contacts.Worker();
      const contacts: IContact[] = await contactsWorker.listContacts();
      inResponse.json(contacts); // serialize object into JSON
    } catch (inError) {
```

```
      inResponse.send("error");
    }
  }
);
```

◇ Register path and method for the endpoint that is used to **add a contact** to the contacts list resource.

```
app.post("/contacts",
  async (inRequest: Request, inResponse: Response) => {
    try {
      const contactsWorker: Contacts.Worker = new Contacts.Worker();
      const contact: IContact = await contactsWorker.addContact (inRequest.body);
      inResponse.json(contact); // for client acknowledgment and future use (includes
          ID)
    } catch (inError) {
      inResponse.send("error");
    }
  }
);
```

◇ Register path and method for the endpoint where the a specific **contact resource** can be **deleted**.

```
app.delete("/contacts/:id",
  async (inRequest: Request, inResponse: Response) => {
    try {
      const contactsWorker: Contacts.Worker = new Contacts.Worker();
      await contactsWorker.deleteContact(inRequest.params.id);
      inResponse.send("ok");
    } catch (inError) {
      inResponse.send("error");
    }
  }
);
```

Part IV: `serverInfo.ts`, to provide info about servers

The Worker classes for IMAP and SMTP will be talking to servers. Therefore, it is necessary to make server information available to them, which is stored in `server/serverInfo.json`.

```
{
  "smtp" : {
    "host" : "mail.mydomain.com", "port" : 999,
    "auth" : {"user" : "user@domain.com", "pass" : "yourpass" } }
}
```

Then proceed with the following code:

◇ Import of modules `path` and `fs` (core Node modules).

```
import path from "path";
import fs from "fs";
```

◇ Export an interface type, which mimics the `server/serverInfo.json` file, and typed variable to store an object that adheres to that interface.

```
export interface IServerInfo {
  smtp : {
    host: string, port: number,
    auth: { user: string, pass: string }
  }
}

export let serverInfo: IServerInfo;
```

◇ Read the `server/serverInfo.json` file and initialize the variable previously created.

```
const rawInfo: string = fs.readFileSync(path.join(__dirname, "../server/serverInfo.json
    "),'utf8');
serverInfo = JSON.parse(rawInfo); // string to object
```

Part V: `SMTP.ts`, to be able to send emails

This file will include class `SMTP.Worker`, accessed in `main.ts`, and uses `nodemailer` module to do the work. This should include the following:

◇ Imports related to the `nodemailer`.

```
import * as nodemailer from "nodemailer";
import { SendMailOptions, SentMessageInfo } from "nodemailer";
import Mail from "nodemailer/lib/mailer";
```

◇ Import the `IServerInfo` from the `serverInfo.ts`.

```
import { IServerInfo } from "./ServerInfo";
```

◇ Define the Worker class, which receives an IServerInfo object and places it in a private member variable, and has a `sendMessage` method.

```
export class Worker {
  private static serverInfo: IServerInfo;
  constructor(inServerInfo: IServerInfo) {
    Worker.serverInfo = inServerInfo;
  }
  public sendMessage(inOptions: SendMailOptions): Promise<void> {
    return new Promise((inResolve, inReject) => {
      const transport: Mail = nodemailer.createTransport(Worker.serverInfo.smtp);
      transport.sendMail(inOptions,
        (inError: Error | null, inInfo: SentMessageInfo) => {
          if (inError) {
            inReject(inError);
          } else {
            inResolve();
          }
        }
      );
    });
  }
}
```

Part VI: `contacts.ts`, for the implementation of contacts functionality

This file includes all the contacts functionality, which will require storage. The NoSQL NeDB API will be used because it has no outside dependencies (no additional server needs to be installed) and simple data will be stored.

◇ Required imports.

```
import * as path from "path";
const Datastore = require("nedb");
```

◇ An interface to describe a contact is required for add/list/delete contact operations. When adding a contact, NeDB will populate the `_id`, so it must be optional.

```
export interface IContact {
  _id?: number, name: string, email: string
}
```

◇ Define the Worker class, which upon construction creates a NeDB Datastore object at the given path. The NeDB creates the file, if it does not exist, or loads it automatically. The Worker class has methods `listContacts`, `addContat` and `deleteContact`.

```
export class Worker {
  private db: Nedb;
  constructor() {
    this.db = new Datastore({
      filename : path.join(__dirname, "contacts.db"),
```

```
      autoload : true
    });
  }

  public listContacts(): Promise<IContact[]> {
    return new Promise((inResolve, inReject) => {
      this.db.find({ },
        (inError: Error | null, inDocs: IContact[]) => {
          if (inError) {
            inReject(inError);
          } else {
            inResolve(inDocs);
          }
        }
      );
    });
  }

  public addContact(inContact: IContact): Promise<IContact> {
    return new Promise((inResolve, inReject) => {
      this.db.insert(inContact,
        (inError: Error | null, inNewDoc: IContact) => {
          if (inError) {
            inReject(inError);
          } else {
            inResolve(inNewDoc);
          }
        }
      );
    });
  }

  public deleteContact(inID: string): Promise<void> {
    return new Promise((inResolve, inReject) => {
      this.db.remove({ _id : inID }, { },
        (inError: Error | null, inNumRemoved: number) => {
          if (inError) {
            inReject(inError);
          } else {
            inResolve();
          }
        }
      );
    });
  }
}
```

Part VII: testing the server side

- ◇ Put everything together and test the server side using the command line tool `https://curl.se/` or Postman, an API development environment `https://www.postman.com/`. Perform tests for endpoints, make screenshots and comment.

Deliver your report!

References

[1] Frank Zammetti, “Modern Full-Stack Development”, 2020.