Creating a full stack application

Future Factory – ZZPP0920

Justus Hänninen – AB6225,

T-03 Gang De Farine 2023

est. 2023

GANG DE FARINE

Table of contents

1 The Plan

As with anything even remotely difficult, you need a plan. What webserver to use, what framework to use, what database solution to use, who develops the backend or the frontend, etc. Deciding these things gives clarity to the project, and when established early in the project, it gives opportunity for anyone in the group to learn from the early stages. These should ideally be decided before a single line of code is written!

1.1 Frontend – Frameworks?

Deciding on a framework can be tough. Sometimes you don't need one, and it might be fun to try and build your own as well! I do heavily recommend learning to use at least one however, as the skills you learn with one framework are somewhat transferable to other frameworks. The few popular frameworks I recommend are React, Angular and Vue.js, in descending order of both popularity (in the job market) and difficulty. So, React is the most difficult but most popular in the job market, and Vue.js is the least popular but the easiest. Your milage may vary on the difficulty part, of course. There are hundreds of frameworks out there, so you won't be wanting for choice!

1.2 Backend – Webservers?

There are only three main webserver solutions. Apache, Nginx and Express. I've used all three personally, and my personal recommendation is Nginx. Our group used Express for the project, but as a webserver, Express is in its adolescence still as of writing. As a benefit however, Express is relatively easy to learn if you know JavaScript since Express works in Node.JS, a JavaScript server solution. Nginx is very feature rich, mature, and very popular in the job market.

1.3 Backend – Databases?

There are multiple popular database solutions, such as PostgreSQL, MariaDB and MongoDB. PostgreSQL is my recommendation, as it is most popular in the job market, and is modern feature wise. If you wish to use JavaScript in your entire stack, then MongoDB is a solid choice as well. MongoDB is relatively volatile however, with features being added and removed constantly as it is very new to the database game.

1.4 Sidenotes

I HEAVILY recommend that you learn TypeScript – AKA JavaScript with types – and this course is the perfect opportunity to do so! JavaScript is one of the more hated languages in the code hobbyist community due to the weird bugs and behaviour caused by being a type-optional language. TypeScript gives reliability to your code, it's comfortable to use once you learn it, and you won't want to go back!

Consider the following JavaScript code:

```
const num1 = 5;
const num2 = "10";
const sum = num1 + num2;
console.log(sum);
```

Many of you will realize that this results in the string “510”, without giving an error. With the following TypeScript changes, our editor will warn us that this is invalid! There is also a compilation process with TypeScript, that will fail in the following case.

```
const num1: number = 5;
const num2: string = "10";
const sum: number = num1 + num2;
console.log(sum);
```

For the purposes of the guide, we will be using NodeJS with Express webserver for the backend, PostgreSQL as the Database and Vue.js (with TypeScript) in the frontend. Another assumption is that whatever server you are using, it’s Linux based. Finally, full stack development now days almost forces you to use NPM to gain access to the gigantic library of… libraries. There are other alternatives to specifically NPM, such as Yarn (which I recommend over base NPM). I won’t give installation instructions to any specific package manager here, since if I had to go step-by-step with even the most basic stuff, this document would rival the bible in length.

2 Know your needs

What is your data? How much of is there? How much of that data do you need in the frontend? Maybe your user interface requires data from the database to render. These questions can be very difficult to answer in the very early stages of the project. You probably have at least some ideas however, through the user stories and feature requirements of your project.

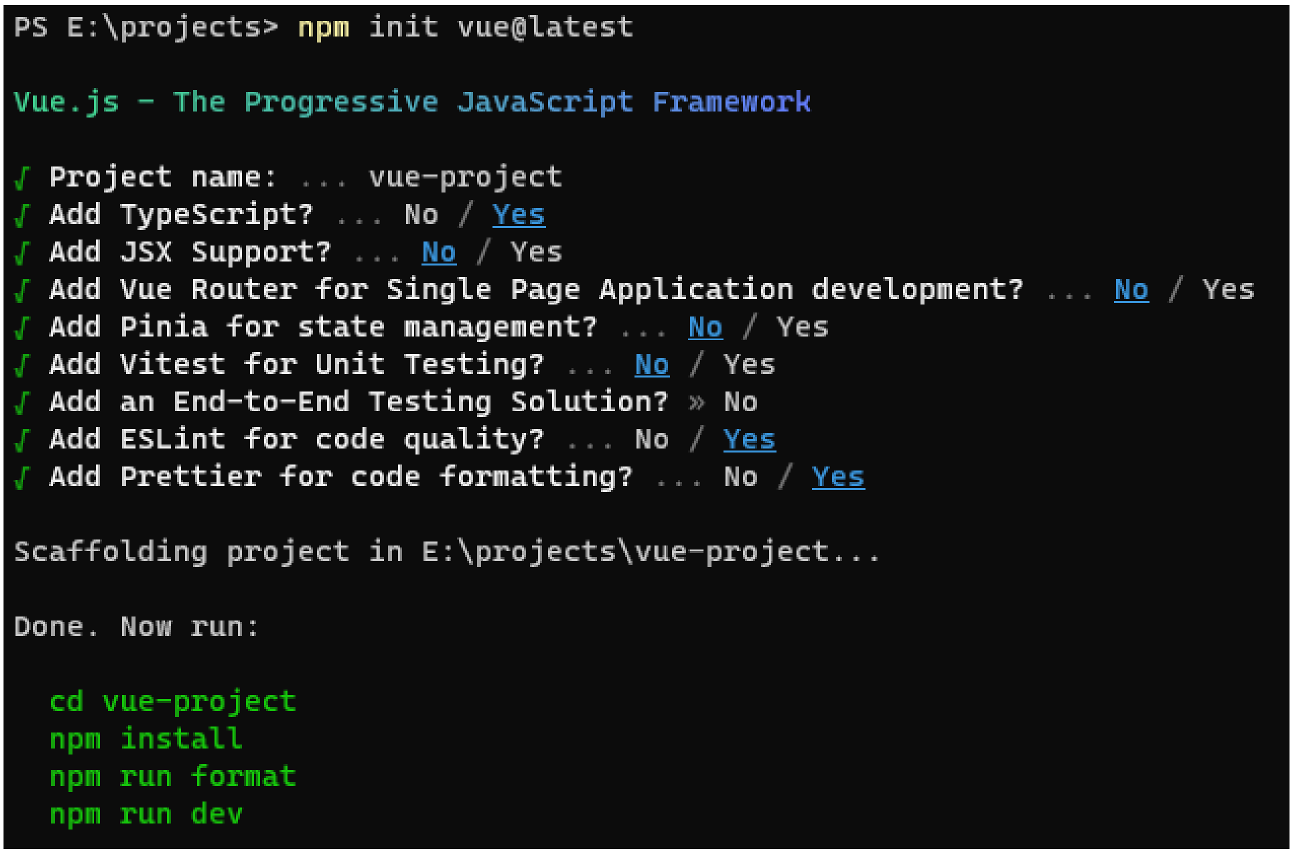
In the Skill Collector project of Future Factory 2022, we built what is basically a glorified questionnaire. This obviously would require saving answers to the database. Another feature requirement was that the questionnaire was to be dynamic. If you changed the database, the questions in the application would reflect that. So, we need a database for that. This type of outline should give enough direction for building your stack.

3 The practical stuff

Once you have an outline for your entire stack, it's time to start building! There is no standard to from which direction it's better to start from. My recommendation is to look at the planned size of your frontend vs. the backend. For our project, the frontend was significantly larger, so we started from there.

3.1 Building a frontend

Like mentioned before, our group used Vue.js. This was just because I as the main developer had quite a bit of experience with it. Most frameworks offer a ready to go installation script through NPM. Just a quick example with Vue.js.



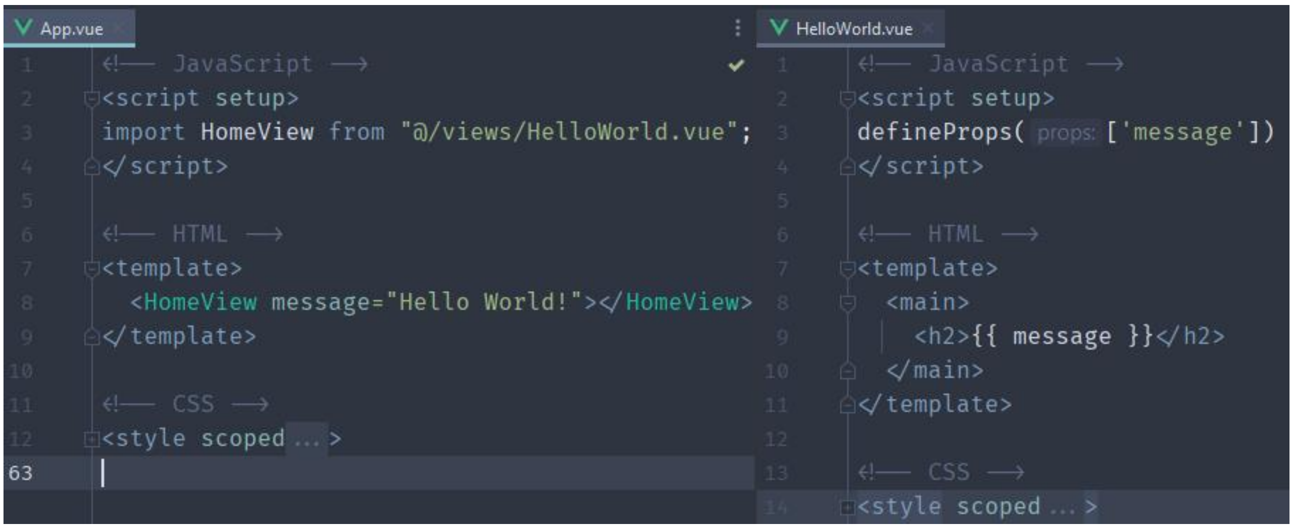
```
PS E:\projects> npm init vue@latest

Vue.js - The Progressive JavaScript Framework

√ Project name: ... vue-project
√ Add TypeScript? ... No / Yes
√ Add JSX Support? ... No / Yes
√ Add Vue Router for Single Page Application development? ... No / Yes
√ Add Pinia for state management? ... No / Yes
√ Add Vitest for Unit Testing? ... No / Yes
√ Add an End-to-End Testing Solution? » No
√ Add ESLint for code quality? ... No / Yes
√ Add Prettier for code formatting? ... No / Yes

Scaffolding project in E:\projects\vue-project...

Done. Now run:

  cd vue-project
  npm install
  npm run format
  npm run dev
```

These guide you through the basic installation, offering quick installs to recommended libraries to different features, such as state management, routing, and using TypeScript (for the love of God use TypeScript). These are not necessary to install here and now, and you probably won't need most of them either. Few recommendations to install at least is – obviously – TypeScript, a linter and prettier. A linter and prettier automatically format your code to adhere to some set guidelines, and when installed through the frame-work's installation script, they come setup properly for your selected framework.

3.1.1 The barebones

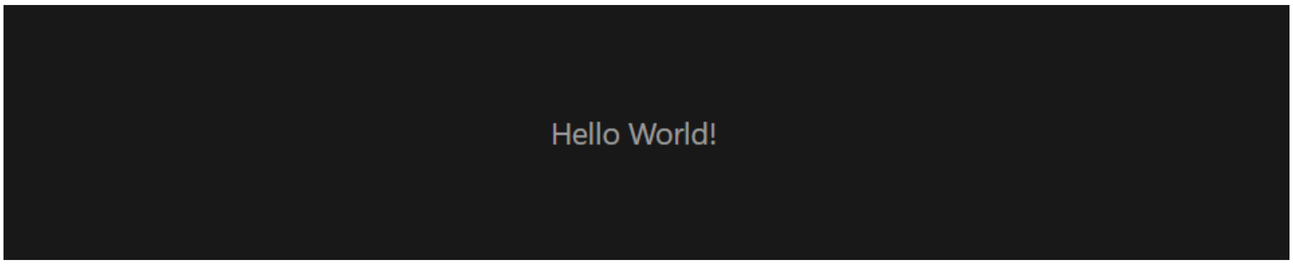
Building a basic application is easy with any framework, and most follow the same mechanics. You have components, and you build your views with these components. You can pass down props from higher up in the component tree, but sending data up the component tree is different for each framework. In Vue we would use emits with custom events, and in React you would pass callable methods. State management is also an option, which allows you store data to be available globally (global as in from any component in the application).

Here's an example of a barebones hello world Vue application, with the main App.vue component passing the string “Hello World!” as a prop to the child component to be displayed.



```
<!-- JavaScript -->
<script setup>
import HomeView from "@/views/HelloWorld.vue";
</script>

<!-- HTML -->
<template>
  <HomeView message="Hello World!"></HomeView>
</template>

<!-- CSS -->
<style scoped ... >
```

```
<!-- JavaScript -->
<script setup>
defineProps( props: ['message'])
</script>

<!-- HTML -->
<template>
  <main>
    <h2>{{ message }}</h2>
  </main>
</template>

<!-- CSS -->
<style scoped ... >
```



Hello World!

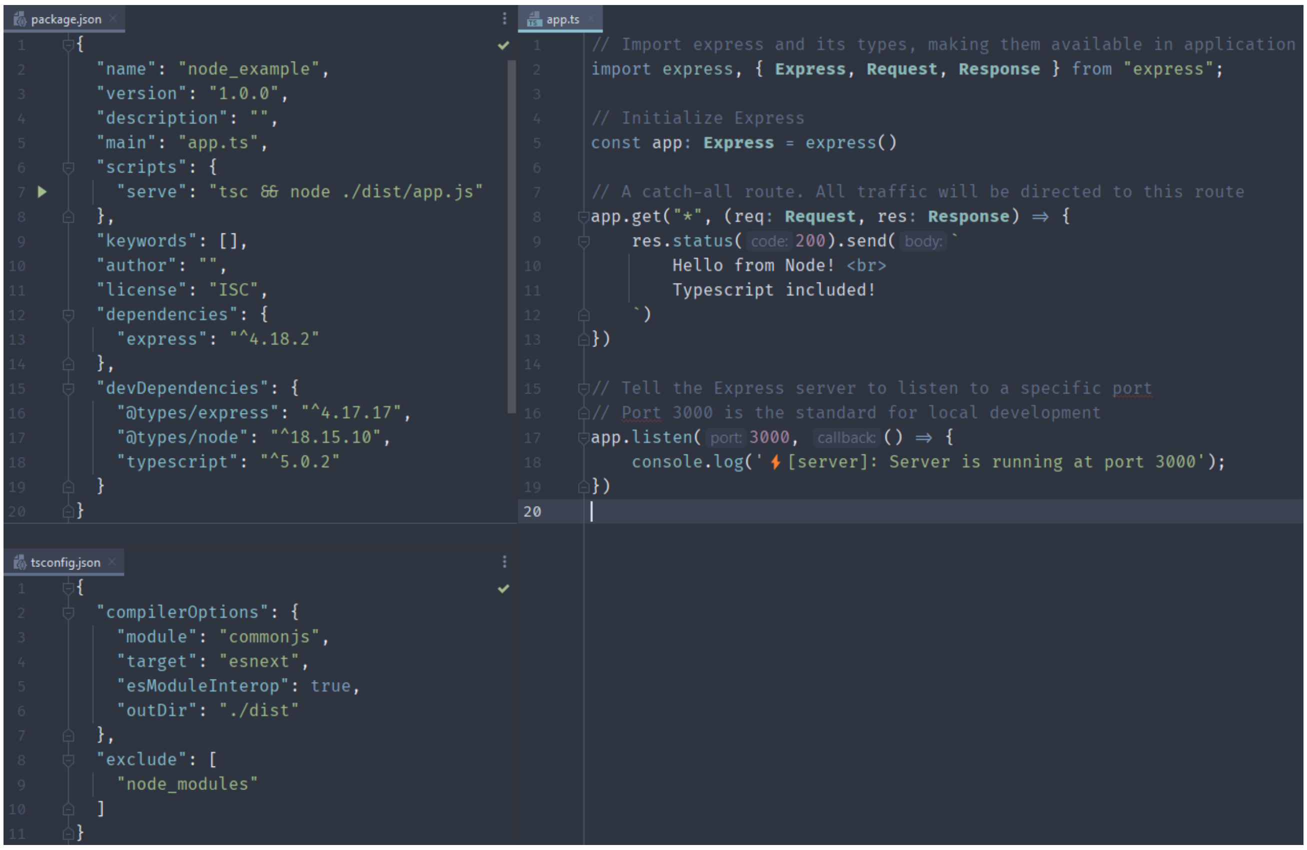
3.2 Building a backend

Again, like Vue.js, our group picked NodeJS with Express just because of prior experience. Your server needs to have NodeJS installed. This also conveniently installs npm. I also recommend installing pm2, as this allows your server to run your NodeJS program as a background service. Follow your server’s operation system specific installation instruction for these. You should by now in your studies know how to create a basic NodeJS/Express application, but just to recap…

To create a basic Node application, all you need to do is run `npm init`. It runs you through a guided process to create a very barebones package.json file. Alongside this you need to manually create a root file – which by default in the package.json is index.js. Node comes with multiple required libraries by default, such as http (or https), but something like Express or Typescript must be installed from NPM. Remember to also install the type library for node from `@types/node`.

Note that Node does not support TypeScript files, so you always have to compile your TypeScript to regular JavaScript before you can run it. You could do this with NPM scripts automatically very simply by changing your package.json file.

Following is a bare minimum configuration to grab send something from your server with NodeJS with TypeScript. Running the command `npm serve` will automatically compile the TypeScript and run the server.



package.json

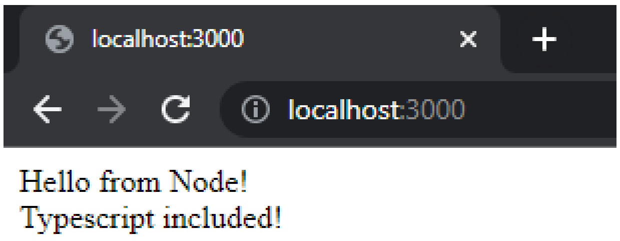
```json
{
  "name": "node_example",
  "version": "1.0.0",
  "description": "",
  "main": "app.ts",
  "scripts": {
    "serve": "tsc && node ./dist/app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "@types/express": "^4.17.17",
    "@types/node": "^18.15.10",
    "typescript": "^5.0.2"
  }
}
```

tsconfig.json

```json
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "esnext",
    "esModuleInterop": true,
    "outDir": "./dist"
  },
  "exclude": [
    "node_modules"
  ]
}
```

app.ts

```typescript
// Import express and its types, making them available in application
import express, { Express, Request, Response } from "express";

// Initialize Express
const app: Express = express()

// A catch-all route. All traffic will be directed to this route
app.get("*", (req: Request, res: Response) => {
    res.status(200).send(`
        Hello from Node! <br>
        Typescript included!
    `)
})

// Tell the Express server to listen to a specific port
// Port 3000 is the standard for local development
app.listen(3000, () => {
    console.log('⚡[server]: Server is running at port 3000');
})
```



localhost:3000

Hello from Node!
Typescript included!

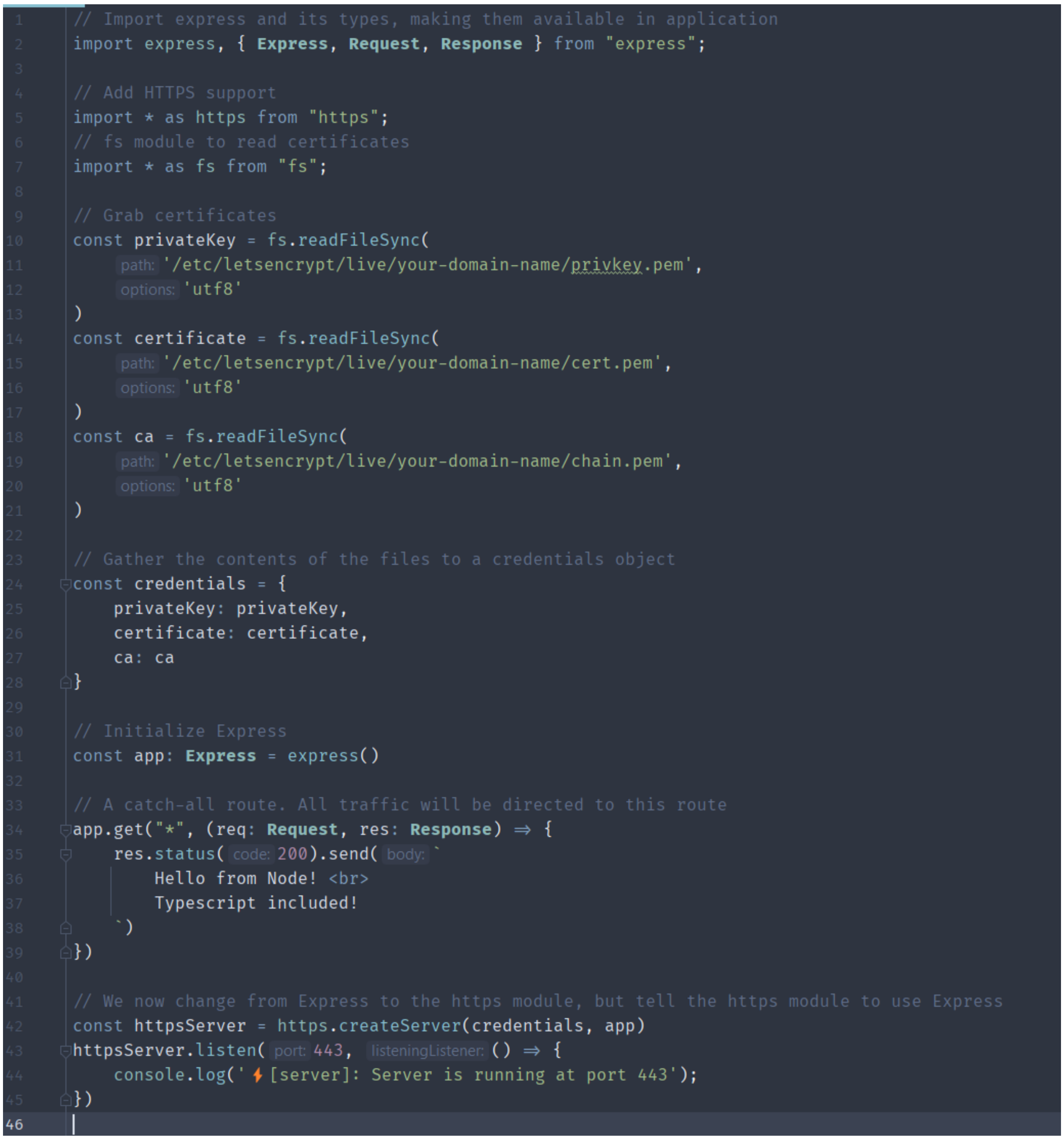
3.2.1 Route specifics

In the picture, you might notice we use app.get for our route. The “get” here is the HTTP method you should be familiar with by now from other courses. Other important ones are “post”, “put”, “update” and “delete”. The bare minimum config I showed also used a wildcard name, but you should name routes more specifically for just security reasons, such as “/users”. Routes with different methods can share names, meaning `app.get(“/users”)` is separate from `app.put(“/users”)`

3.2.2 Adding HTTPS support

The easiest way to use HTTPS is to install Let’s Encrypt on your server. It will handle both generating, and regenerating the certificates before they expire. On a Linux server you can install the “letsencrypt” package. Generating certificates with Let’s Encrypt are easily found from google, so let’s focus on the important part, using HTTPS with your application.

For Apache and Nginx, there should be commented out configurations that you just uncomment and point to the correct files, but for Express, the process is slightly more involved. We don’t need any third-party libraries for this, thankfully. The following is a simple configuration to enable HTTPS on an Express server

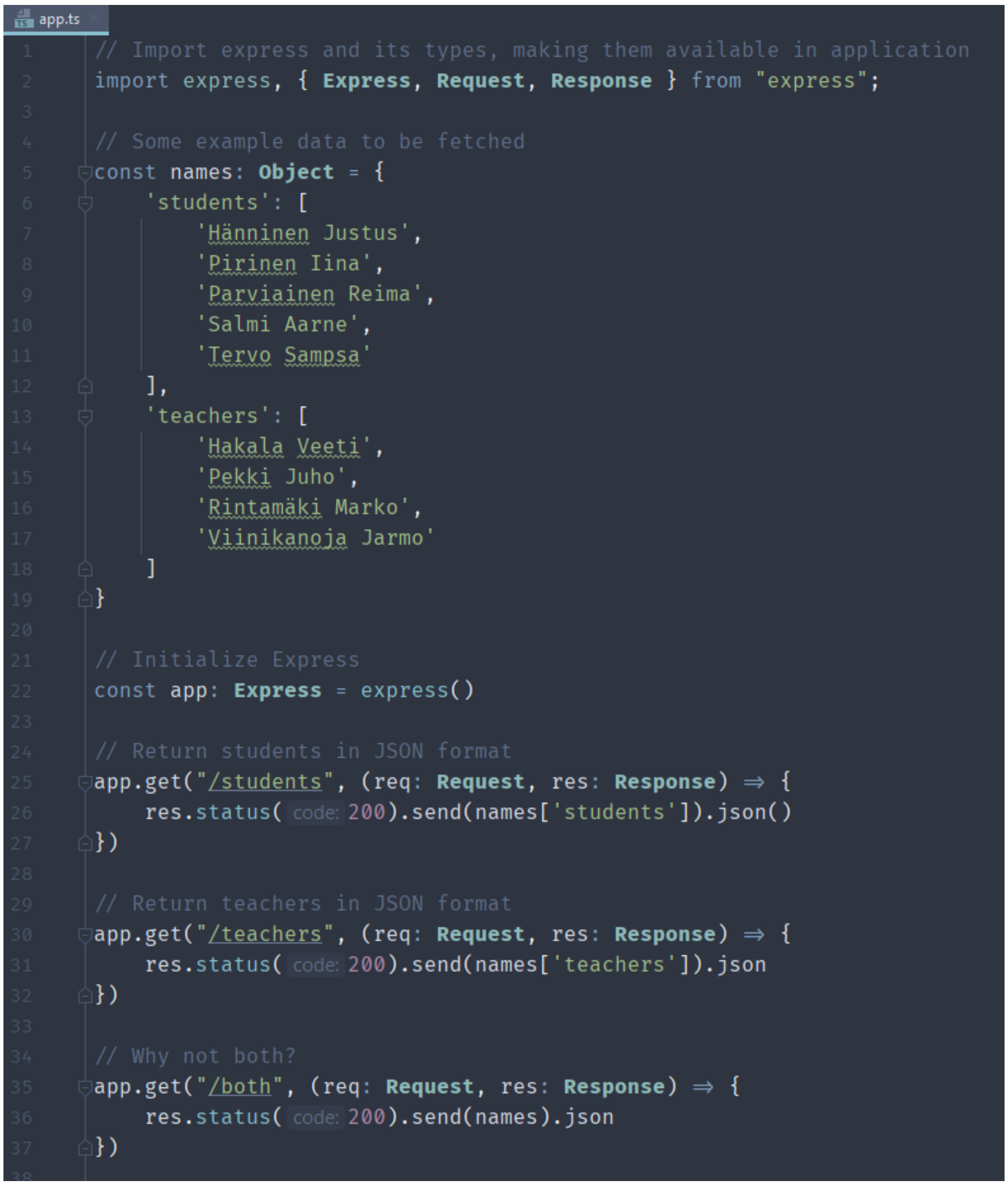


```
// Import express and its types, making them available in application
import express, { Express, Request, Response } from "express";

// Add HTTPS support
import * as https from "https";
// fs module to read certificates
import * as fs from "fs";

// Grab certificates
const privateKey = fs.readFileSync(
    path: '/etc/letsencrypt/live/your-domain-name/privkey.pem',
    options: 'utf8'
)
const certificate = fs.readFileSync(
    path: '/etc/letsencrypt/live/your-domain-name/cert.pem',
    options: 'utf8'
)
const ca = fs.readFileSync(
    path: '/etc/letsencrypt/live/your-domain-name/chain.pem',
    options: 'utf8'
)

// Gather the contents of the files to a credentials object
const credentials = {
    privateKey: privateKey,
    certificate: certificate,
    ca: ca
}

// Initialize Express
const app: Express = express()

// A catch-all route. All traffic will be directed to this route
app.get("*", (req: Request, res: Response) => {
    res.status( code: 200).send( body: `
        Hello from Node! <br>
        Typescript included!
    `)
})

// We now change from Express to the https module, but tell the https module to use Express
const httpsServer = https.createServer(credentials, app)
httpsServer.listen( port: 443,  listeningListener: () => {
    console.log('⚡[server]: Server is running at port 443');
})
```

As a bonus, to ease development, you could add the “dotenv” library. With this, through a conditional that reads the environment variables, you could use regular Express for local development, or enable the HTTPS configuration on the server.

3.3 Connecting your backend to your frontend

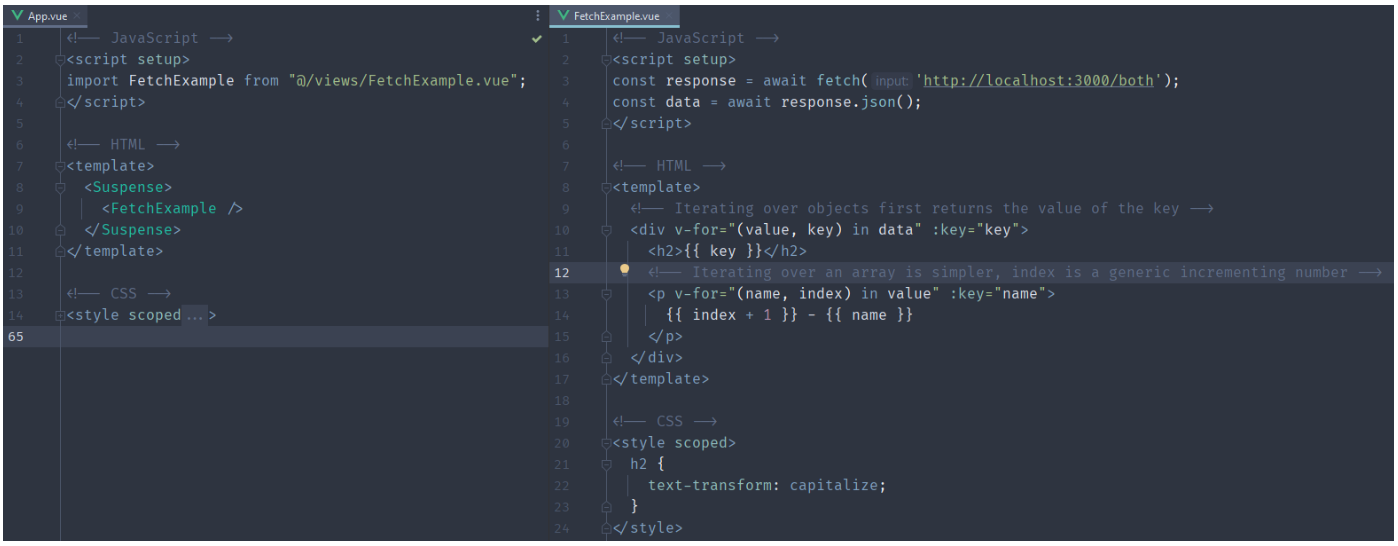
For the sake of the demo, I didn’t want to build a database and connect it to the backend. I will leave that as an exercise to the reader. It’s very simple stuff, and NodeJS for example has libraries that give you easy to use functions for most (if not all) existing databases, such as “mongoose” for MongoDB or “pg” for PostgreSQL. Let’s modify our backend to return data from some routes!



app.ts

```
// Import express and its types, making them available in application
import express, { Express, Request, Response } from "express";

// Some example data to be fetched
const names: Object = {
    'students': [
        'Hänninen Justus',
        'Pirinen Iina',
        'Parviainen Reima',
        'Salmi Aarne',
        'Tervo Sampsa'
    ],
    'teachers': [
        'Hakala Veeti',
        'Pekki Juho',
        'Rintamäki Marko',
        'Viinikanoja Jarmo'
    ]
}

// Initialize Express
const app: Express = express()

// Return students in JSON format
app.get("/students", (req: Request, res: Response) ⇒ {
    res.status( code: 200).send(names['students']).json()
})

// Return teachers in JSON format
app.get("/teachers", (req: Request, res: Response) ⇒ {
    res.status( code: 200).send(names['teachers']).json
})

// Why not both?
app.get("/both", (req: Request, res: Response) ⇒ {
    res.status( code: 200).send(names).json
})
```

Getting data from the backend should be familiar if you've completed the second-year full stack course. To recap, we can use the native fetch command. The following example will fetch data from the "/both" route and render each key and array in a neat list using Vue's v-for method. Note the "Suspense" tag around our "FetchExample" component. This is required in Vue when the component has asynchronous functions in its setup.

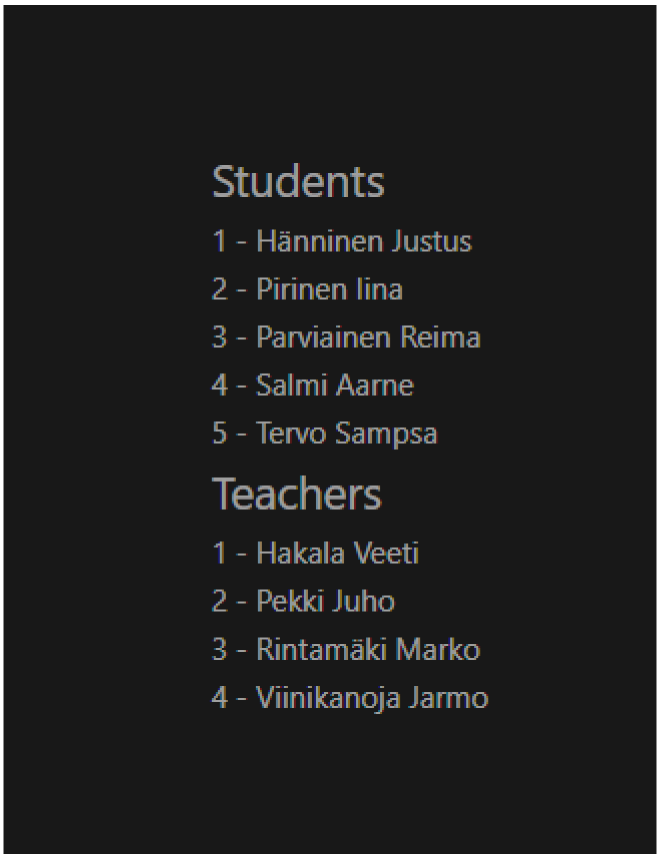


App.vue

```
<!-- JavaScript -->
<script setup>
import FetchExample from "@/views/FetchExample.vue";
</script>

<!-- HTML -->
<template>
  <Suspense>
    <FetchExample />
  </Suspense>
</template>

<!-- CSS -->
<style scoped...>
```

FetchExample.vue

```
<!-- JavaScript -->
<script setup>
const response = await fetch('http://localhost:3000/both');
const data = await response.json();
</script>

<!-- HTML -->
<template>
  <!-- Iterating over objects first returns the value of the key -->
  <div v-for="(value, key) in data" :key="key">
    <h2>{{ key }}</h2>
    <!-- Iterating over an array is simpler, index is a generic incrementing number -->
    <p v-for="(name, index) in value" :key="name">
      {{ index + 1 }} - {{ name }}
    </p>
  </div>
</template>

<!-- CSS -->
<style scoped>
  h2 {
    text-transform: capitalize;
  }
</style>
```



```
Students
1 - Hänninen Justus
2 - Pirinen Iina
3 - Parviainen Reima
4 - Salmi Aarne
5 - Tervo Sampsa
Teachers
1 - Hakala Veeti
2 - Pekki Juho
3 - Rintamäki Marko
4 - Viinikanoja Jarmo
```

Happy Coding!