Exercise 3

Understanding more about creating good RESTful APIs in Typescript

Prior Knowledge
Previous exercise

Objectives
Understanding the basics of Typescript & Node
Controllers, Routes, Models
Building Typescript and testing

Software Requirements
Yarn, Git, vscode, etc

Steps

1. I have created a sample program that you can view and edit. It shows a better designed approach than the last program. In this approach we have a "model, controller, routing" type approach for creating RESTful APIs.

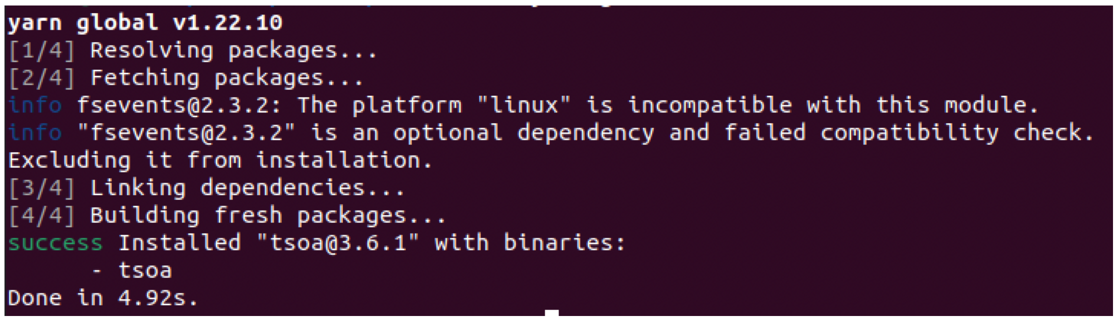
2. The app is a simple "database" of phones. In fact there is no database. Instead I have used the "singleton pattern" (https://en.wikipedia.org/wiki/Singleton_pattern) to mock up a database. In a later exercise we will add a genuine database and show how we can package everything up to work nicely.

3. In addition to express, this project also uses a project called "tsoa":
   https://github.com/lukeautry/tsoa
   https://tsoa-community.github.io/docs/

   tsoa automates a bunch of aspects of building RESTful APIs in Typescript - especially Swagger (which is coming later). However, we might as well start as we wish to go on.

4. Let’s install tsoa globally (since it has a CLI):

```
yarn global add tsoa
```



```
yarn global v1.22.10
[1/4] Resolving packages...
[2/4] Fetching packages...
info fsevents@2.3.2: The platform "linux" is incompatible with this module.
info "fsevents@2.3.2" is an optional dependency and failed compatibility check.
Excluding it from installation.
[3/4] Linking dependencies...
[4/4] Building fresh packages...
success Installed "tsoa@3.6.1" with binaries:
      - tsoa
Done in 4.92s.
```

5. Let’s clone the git repo:

```
cd ~
git clone https://github.com/pzfreo/ts-express-api-starter.git
cd ts-express-api-starter
```

6. Let's install a useful utility called tree:

   ```
   sudo apt install tree
   ```

7. Now run it:

   ```
   tree
   ```

   You should see:

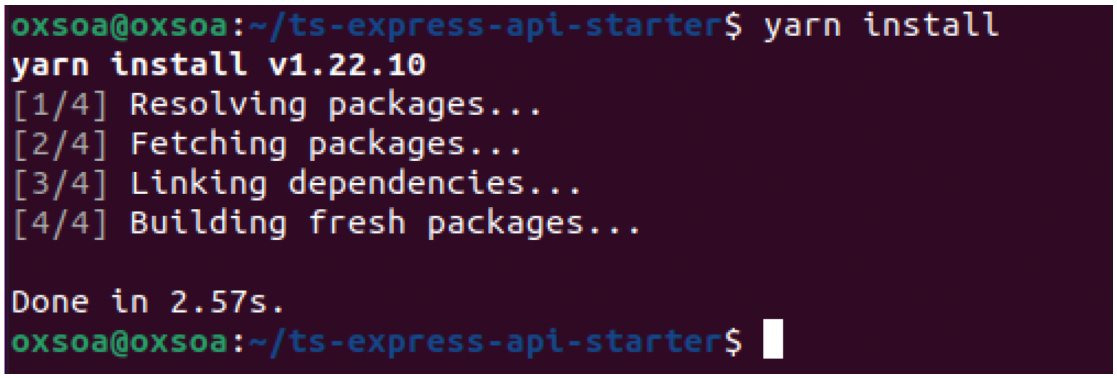
   ```
   .
   ├── LICENSE
   ├── README.md
   ├── package.json
   ├── phone-postman.json
   ├── second.json
   ├── src
   │   ├── Phone.ts
   │   ├── PhoneController.ts
   │   ├── PhoneService.ts
   │   ├── app.ts
   │   └── generated
   │       └── routes.ts
   ├── tsconfig.json
   ├── tsoa.json
   └── yarn.lock

   2 directories, 13 files
   ```

   You should be familiar with the overall layout. The new things are:

Phone.ts	This is a model that defines our "Phone" objects/types
PhoneService.ts	This is the "backend" that abstracts away from the database and gives us access to a table of phones
PhoneController.ts	This is where we define how to handle GET/PUT/POST etc. Effectively this bridges between the HTTP requests and the Service
generated/ routes.ts	This is some clever logic that actually is auto-generated by tsoa to make express call the controller methods

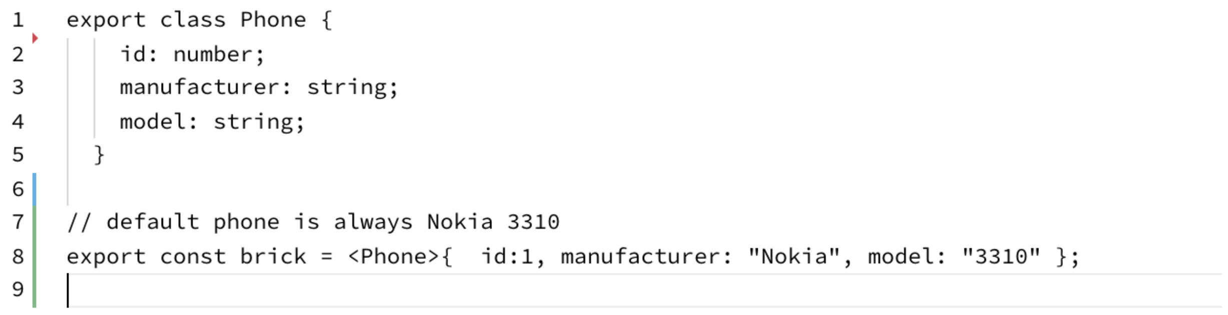
8. Before we start coding, we need to install the dependencies that are specified in package.json

```
yarn install
```



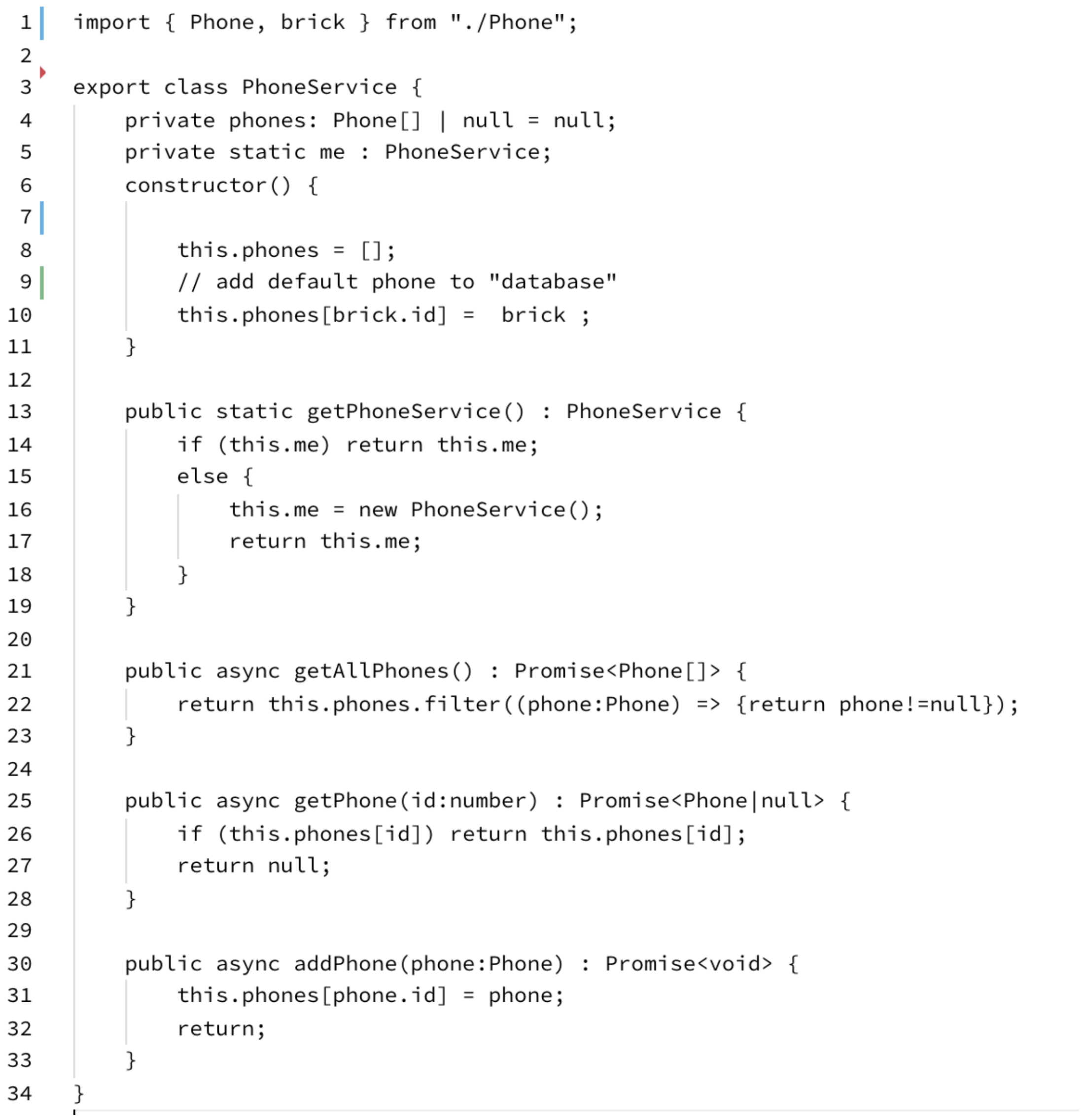
```
oxsoa@oxsoa:~/ts-express-api-starter$ yarn install
yarn install v1.22.10
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

Done in 2.57s.
oxsoa@oxsoa:~/ts-express-api-starter$
```

9. Let’s look at the model first - **Phone.ts.** This is really simple:



```
export class Phone {
    id: number;
    manufacturer: string;
    model: string;
  }

// default phone is always Nokia 3310
export const brick = <Phone>{  id:1, manufacturer: "Nokia", model: "3310" };

```

10. Now let’s look at the PhoneService.ts:



```
 1  import { Phone, brick } from "./Phone";
 2
 3  export class PhoneService {
 4      private phones: Phone[] | null = null;
 5      private static me : PhoneService;
 6      constructor() {
 7
 8          this.phones = [];
 9          // add default phone to "database"
10          this.phones[brick.id] =  brick ;
11      }
12
13      public static getPhoneService() : PhoneService {
14          if (this.me) return this.me;
15          else {
16              this.me = new PhoneService();
17              return this.me;
18          }
19      }
20
21      public async getAllPhones() : Promise<Phone[]> {
22          return this.phones.filter((phone:Phone) => {return phone!=null});
23      }
24
25      public async getPhone(id:number) : Promise<Phone|null> {
26          if (this.phones[id]) return this.phones[id];
27          return null;
28      }
29
30      public async addPhone(phone:Phone) : Promise<void> {
31          this.phones[phone.id] = phone;
32          return;
33      }
34  }
```

Lines 1-19 are basically setting up a default array in memory with one Phone pre-defined.

Lines 21-34 are the actual logic.

The interesting aspect of this is that we have defined all this in an async way, where each method is defined as an async function, and returns a Promise with a specific type.

The Promise is a clever approach in JavaScript/Typescript that lets the whole service be highly asynchronous. When we add a database this really enhances the scalability and robustness of the code compared to a threaded model.

Please note that you don’t need to fully understand Promises and node’s async architecture to learn the main objectives of this course (how to structure REST APIs, etc). Hence why I have created starter code to help out so we can focus on the core learning objectives and not general coding.

If you do want to read more about Promises, you can read here:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

11. Now the really interesting code is the **PhoneController.ts**:

```
import {
  Controller,
  Get,
  Post,
  Path,
  Route,
  Body
 } from "tsoa";
 import { Phone } from "./Phone";
 import { PhoneService} from "./PhoneService";

 @Route("/phones")
 export class PhoneController extends Controller {
  @Get("/")
  public async getPhones() : Promise<Phone[]> {
    const phoneService: PhoneService = PhoneService.getPhoneService();
    return await phoneService.getAllPhones();
  }
  @Post("/")
  public async createPhone(
    @Body() phone: Phone
  ) : Promise<Phone> {
    const phoneService: PhoneService = PhoneService.getPhoneService();
    await phoneService.addPhone(phone);
    // send back what is in the database in case there is logic that changes it on saving
    return await phoneService.getPhone(phone.id);
  }

 }
```

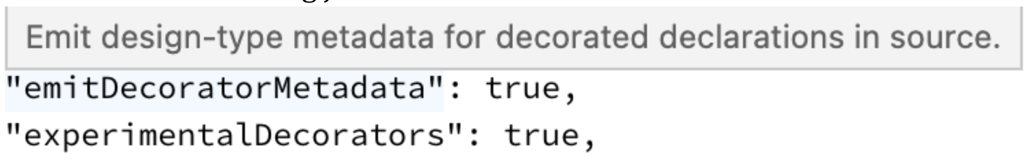
It also uses Promises and async functions.
There is a new keyword you will see here: "await"

Basically this ensures that the async function completes at this point. For example if you want to evaluate the value of:

`phoneService.getPhone(phone.id)`

in your logic, then you need to "await" it otherwise its still just a Promise to get that result at some point in the future.

12. The first main thing to notice are the "decorators" (starting with @). These are based on a new capability in typescript which needs to be enabled. Look at tsconfig.json:



Emit design-type metadata for decorated declarations in source.

```
"emitDecoratorMetadata": true,
"experimentalDecorators": true,
```

13. These decorators are from the tsoa package and they really help us build more readable and simple logic for defining RESTful APIs in ts.

14. The first decorator annotates the whole class:

    `@Route("/phone")`

    This tells our code to automatically route any requests from /phones to the methods in this class.

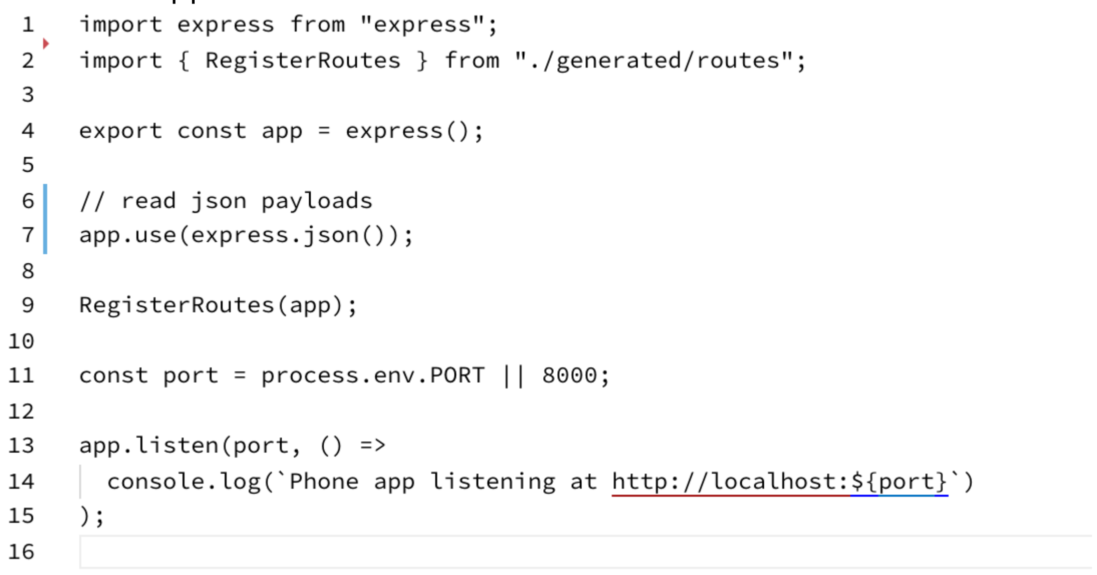
    `@Get("/")`
    This basically says any HTTP calls to "GET /phones" come here. The URL is the combination of the overall route at the top of the class with any specific extra we define in the Get suffix.

    `@Post` is just like `@Get`.

    `@Body` decorates the parameter to say that the POST Body should come here.

15. How does the control flow actually get here?

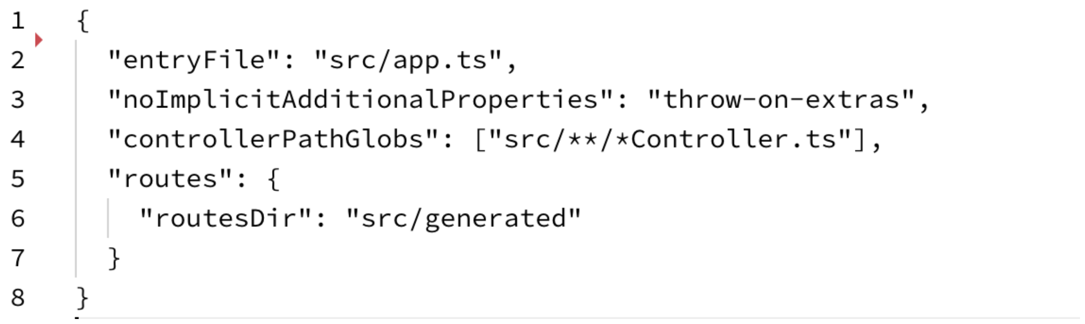
    Let's look at `app.ts`



```
1   import express from "express";
2   import { RegisterRoutes } from "./generated/routes";
3
4   export const app = express();
5
6   // read json payloads
7   app.use(express.json());
8
9   RegisterRoutes(app);
10
11  const port = process.env.PORT || 8000;
12
13  app.listen(port, () =>
14    console.log(`Phone app listening at http://localhost:${port}`)
15  );
16
```

    Line 9 does the hard work, but before it can do that, we need to import the "RegisterRoutes" function at line 2.

16. Now is a good time to take a quick look at the **tsoa.json** config:



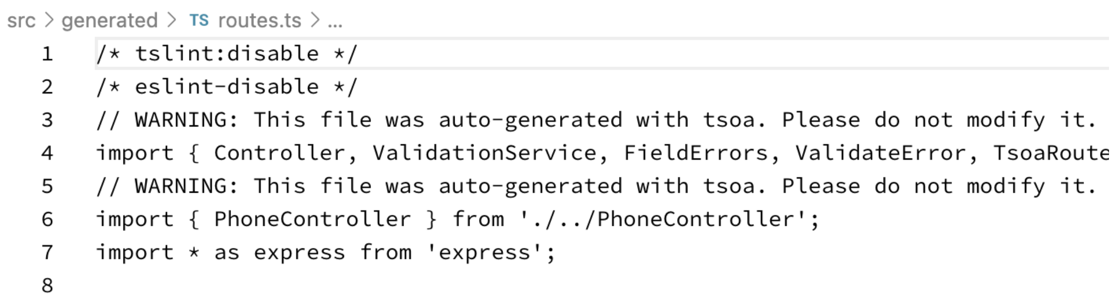
```
{
  "entryFile": "src/app.ts",
  "noImplicitAdditionalProperties": "throw-on-extras",
  "controllerPathGlobs": ["src/**/*Controller.ts"],
  "routes": {
    "routesDir": "src/generated"
  }
}
```

Notice how tsoa is configured to read any controllers in the src directory (and that they **must** be called xxxController.ts to be spotted!). Notice also that I have decided to have a generated/ directory in src where the code lives. I know this is stating the obvious but, this must match the import in app.ts!

17. To run the code generation, type:

```
tsoa routes
```

There is no output on the console, but routes.ts will get generated or updated. Take a quick look at `src/generated/routes.ts`



src > generated > TS routes.ts > ...

```
/* tslint:disable */
/* eslint-disable */
// WARNING: This file was auto-generated with tsoa. Please do not modify it.
import { Controller, ValidationService, FieldErrors, ValidateError, TsoaRoute
// WARNING: This file was auto-generated with tsoa. Please do not modify it.
import { PhoneController } from './../PhoneController';
import * as express from 'express';

```

This code is auto-generated from the controller. You can take a look to see roughly what is going on. Don't edit this code (ever), as it gets re-generated every time we change the controller.

18. If you look at the scripts in package.json, you will see that every time we run this we do "tsoa routes" to ensure the code is updated. We exclude the generated code from the watch because it can end up in a loop.

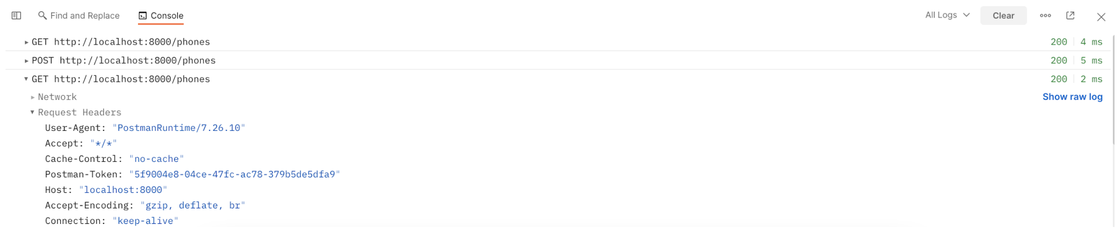
19. There is a collection of tests in `phone-postman.json`

20. After running ‘yarn run dev’, you can run the tests in one of three ways:

    a. `yarn run test-dev`
    b. `newman run phone-postman.json`
    c. Import into Postman and run from the UI

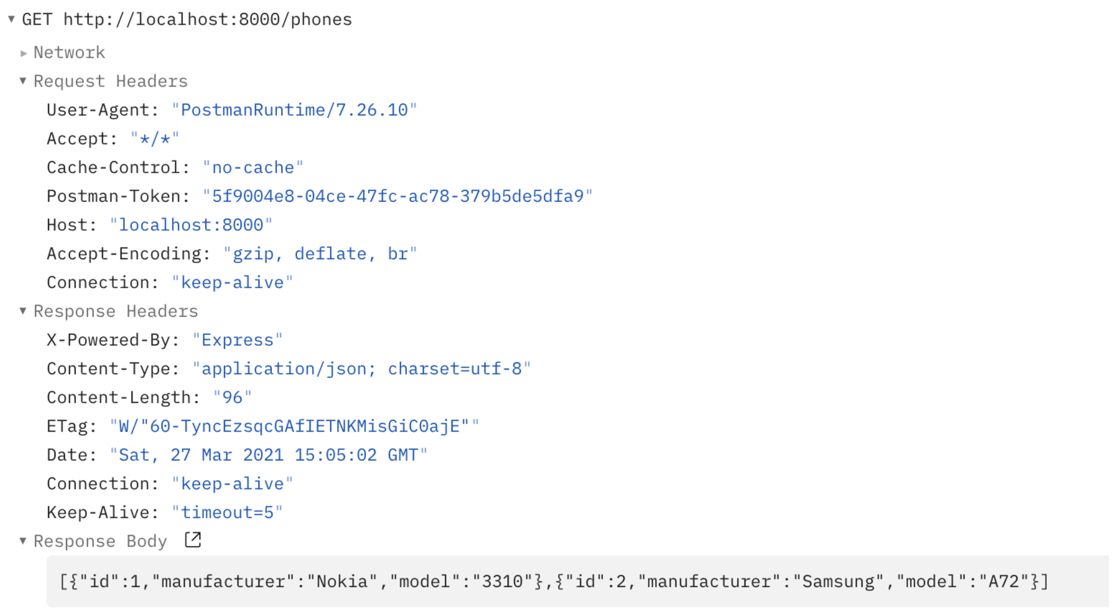
21. Run the tests and see what is going on.

22. Notice that if you expand the “Console” in the Postman UI you can see exactly the HTTP flow in each test:



```
▸ GET http://localhost:8000/phones                    200  4 ms
▸ POST http://localhost:8000/phones                   200  5 ms
▾ GET http://localhost:8000/phones                    200  2 ms
  ▸ Network
  ▾ Request Headers
     User-Agent: "PostmanRuntime/7.26.10"
     Accept: "*/*"
     Cache-Control: "no-cache"
     Postman-Token: "5f9004e8-04ce-47fc-ac78-379b5de5dfa9"
     Host: "localhost:8000"
     Accept-Encoding: "gzip, deflate, br"
     Connection: "keep-alive"
```

And looking more closely:

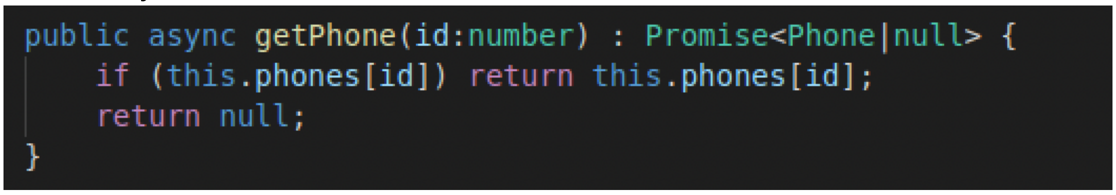


```
▾ GET http://localhost:8000/phones
  ▸ Network
  ▾ Request Headers
     User-Agent: "PostmanRuntime/7.26.10"
     Accept: "*/*"
     Cache-Control: "no-cache"
     Postman-Token: "5f9004e8-04ce-47fc-ac78-379b5de5dfa9"
     Host: "localhost:8000"
     Accept-Encoding: "gzip, deflate, br"
     Connection: "keep-alive"
  ▾ Response Headers
     X-Powered-By: "Express"
     Content-Type: "application/json; charset=utf-8"
     Content-Length: "96"
     ETag: "W/"60-TyncEzsqcGAfIETNKMisGiC0ajE""
     Date: "Sat, 27 Mar 2021 15:05:02 GMT"
     Connection: "keep-alive"
     Keep-Alive: "timeout=5"
  ▾ Response Body
     [{"id":1,"manufacturer":"Nokia","model":"3310"},{"id":2,"manufacturer":"Samsung","model":"A72"}]
```

23. Let’s enhance the code to allow us to find a specific phone by its id.

24. We need to pass the id to the server. A common approach is to embed it into the path. e.g. http://localhost:8000/phones/1

25. I’ve already written the backend service method:



```
public async getPhone(id:number) : Promise<Phone|null> {
    if (this.phones[id]) return this.phones[id];
    return null;
}
```

26. In PhoneController.ts add the following:

    ```
    @Get("/{id}")
    public async getPhone(
        @Path() id: number): Promise<Phone> {
            // implement logic here
    }
    ```

    You will need to fix the imports as well.

    This time the decoration has special characters { }. These indicate that id is a “variable”. The second decoration @Path then links the data from the incoming Path in the HTTP request to the parameter in the method.

    Fill in the logic to return the phone JSON.

27. What is the response if you try to GET a phone id that doesn’t exist? (e.g. GET /phones/999)

    What do you think should happen?

28. There is a second set of tests in second.json

29. Fix up the code so that the second set of tests pass. Look at the test to see what status I’m expecting for a missing id.

    Hint you will need to be able to change the status of the response from within the controller. To do this you can do e.g.:

    ```
    this.setStatus(300);
    ```

30. Congratulations the main lab is done!

Extension

1. Extend the Phone model to include a price. Run the tests and see what happens. Fix the test suite so it works again.

2. Extend the controller and service to enable deleting a Phone by using an HTTP DELETE verb against the URL

   e.g.
   ```
   DELETE /phones/2 HTTP/1.1
   ```

3. That’s all for now