Tech Returners

🧪👾 Welcome to API Lab 1

In this lab, you'll use the ExpressJS library to create a Web API to serve back JSON Responses based on a HTTP Request. You will be using a Test-Driven approach.

ExpressJS is an extremely popular Node-based JavaScript framework which acts as a web server, allowing developers to create many different types of web application. In our case, it will provide the basis for us to create an API.

We will also be using TypeScript for this project 🥳

🛠️ What do I need for this Lab?

You will need:

- Google Chrome Web Browser
- Visual Studio Code
- NodeJS
- Familiarity with TypeScript

1️⃣Create a new node project

We'll make use of Node to create a new project.

👉 Firstly create a new directory to house your code. Let's call the directory the **drinks-service-api:**

```
mkdir drinks-service-api
```

👉 Using terminal navigate to your **drinks-service-api** directory:

```
cd drinks-service-api
```

👉 Whilst in the drinks-service-api directory, initialise the project by running:

```
npm init -f
```

This will create a new Node project ready for us to get started. The `-f` switch is short for "force", which just accepts to the default values of all of the questions `npm` normally asks when we run `npm init`. You should now see a **package.json** in the directory. For example, here is the output of the `ls` command:

```
jamesheggs@Jamess-MBP-2 drinks-service-api % ls
package.json
```

2️⃣ Configure gitignore

We now want to ensure certain files are not included when performing git commits. Specifically it is crucial to ignore any content within the **node_modules** directory. When you install dependencies with node it installs them into the node_modules directory, and we don't want to include those in version control. Any other engineers that clone your repository can run `npm install` to install those dependencies. We can gitignore them

👉 Open up the **drinks-service-api** directory in Visual Studio code

👉 Create a new file called **.gitignore** in the root of your project. It should be the same level as the package.json

👉 Place the following contents into that file

```
node_modules
```

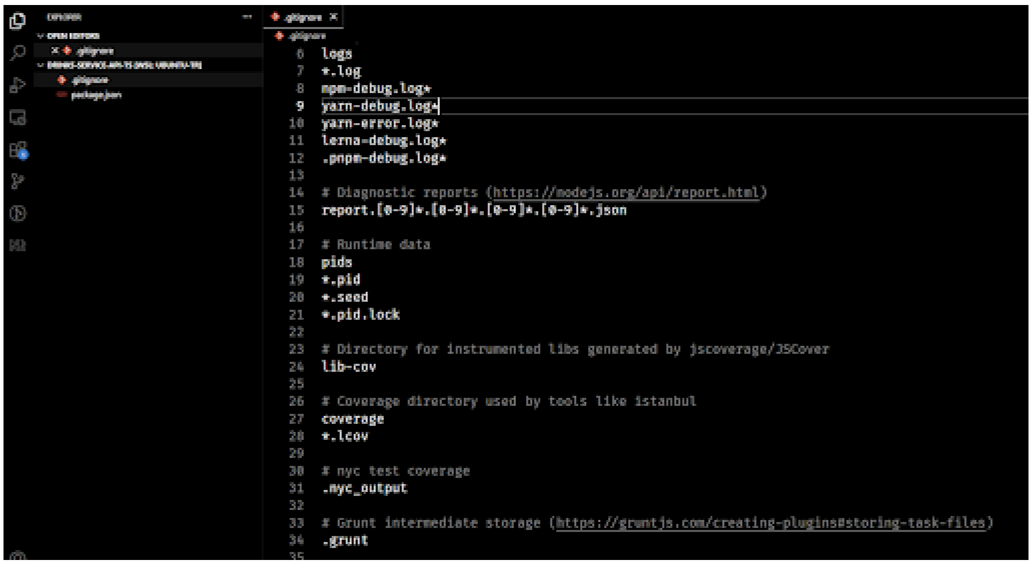
We can do a bit better than that `.gitignore`, though. A quick look at a service called gitignore.io shows that we can generate a more comprehensive `.gitignore` for Node projects:

👉 Open up gitignore.io in your browser.

👉 Add `Node` to the text box and hit enter.

👉 Generate a `.gitignore` file and copy/paste it into your `.gitignore`

At this point your project should look something like the following:



```
logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*
lerna-debug.log*
.pnpm-debug.log*

# Diagnostic reports (https://nodejs.org/api/report.html)
report.[0-9]*.[0-9]*.[0-9]*.[0-9]*.json

# Runtime data
pids
*.pid
*.seed
*.pid.lock

# Directory for instrumented libs generated by jscoverage/JSCover
lib-cov

# Coverage directory used by tools like istanbul
coverage
*.lcov

# nyc test coverage
.nyc_output

# Grunt intermediate storage (https://gruntjs.com/creating-plugins#storing-task-files)
.grunt
```

3️⃣Install dependencies

To create our API we'll make use of some testing frameworks—namely Jest and Supertest—and ExpressJS for handling requests to the API. Of course we'll also need TypeScript and a few other handy tools. Let's install everything now.

> To make full use of TypeScript, it's helpful to have well-defined type information for any dependencies that we consume. Some `npm` libraries include their TypeScript type information by default. Others package their types separately using the prefix `@types`. When installing a new dependency, check the documentation to see where their types are stored and whether you need to install the types along with the package.

👉 Go back to the terminal, making sure you are in the root of your project (where the `package.json` is) and install TypeScript and the tools we'll use to run our code:

```
npm install typescript@4.9.4 ts-node@10.9.1 nodemon@2.0.20
```

👉 Next you can also install Jest and its dependencies

```
npm install -D jest@29.3.1 ts-jest@29.0.3 @types/jest@29.2.5
@types/node@18.11.18
```

👉Next install Supertest and its corresponding `types` package:

```
npm install -D supertest@6.3.3 @types/supertest@2.0.12
```

👉 Finally install Express and its associated types:

```
npm install express@4.18.2
npm install @types/express@4.17.15
```

Here we fix the versions of our dependencies using the @ sign when installing them. This is done for the purposes of ensuring the tutorial works irrespective of which versions are released by those frameworks. If you wanted to simply install the latest versions then you would run **npm install packageName**

Your package.json should now look like this example below:

```
{
     "name": "drinks-service-api-ts",
     "version": "1.0.0",
     "description": "",
     "main": "index.js",
     "scripts": {
          "test": "echo \"Error: no test specified\" && exit 
1"
     },
     "keywords": [],
     "author": "",
     "license": "ISC",
     "devDependencies": {
          "@types/jest": "^29.2.5",
          "@types/node": "^18.11.18",
          "@types/supertest": "^2.0.12",
          "jest": "^29.3.1",
          "supertest": "^6.3.3",
          "ts-jest": "^29.0.3"
     },
     "dependencies": {
          "@types/express": "^4.17.15",
          "express": "^4.18.2",
          "nodemon": "^2.0.20",
          "ts-node": "^10.9.1",
          "typescript": "^4.9.4"
     }
}
```

3️⃣💠5️⃣ Configure Jest and TypeScript

We need to get Jest and TypeScript to play nicely with each other. Jest is a JavaScript framework so by default it will complain if we try to run a TS file through it.

First we run a command that configures Jest to use `ts-jest` to preprocess the files before Jest tries to run them:

👉`npx ts-jest config:init`

👉 Next we need tell node to run the jest command when the test script is invoked. Update the **package.json** so that the **scripts** section goes from looking like this:

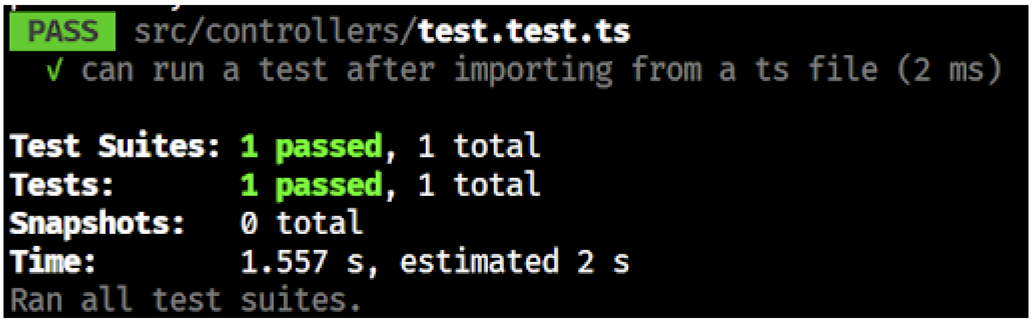
```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
},
```

To look like this:

```
"scripts": {
    "test": "jest"
},
```

💡 We could alter that command to use “watch mode”, or to only load certain test files, or whatever else we like. For now we’ll stick with the default behaviour.

This is enough to create a `.ts` file and a `.test.ts` file which imports from it, and everything should “just work”:



```
 PASS  src/controllers/test.test.ts
  √ can run a test after importing from a ts file (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.557 s, estimated 2 s
Ran all test suites.
```

But we can also add a `tsconfig` file to both be a little more controlled in how TypeScript behaves and also to enable our tools to work together.

👉 Add a file called `tsconfig.json` to the root of your project.

👉 Copy the following into your `tsconfig`:

```
{
      "include": ["./src/**/*"],
      "exclude": ["node_modules"],
      "compilerOptions": {
            "strict": true /* Enable all strict type-checking
options. */,
            "target": "ES2022",
            "module": "ES2015",
            "moduleResolution": "node",
            "rootDir": "./src",
            "outDir": "./build",
            "esModuleInterop": true /* Emit additional
JavaScript to ease support for importing CommonJS modules.
This enables 'allowSyntheticDefaultImports' for type
compatibility. */,
            "forceConsistentCasingInFileNames": true /* Ensure
that casing is correct in imports. */,
            "skipLibCheck": true /* Skip type checking all .d.ts
files. */
      },
      "ts-node": {
            /* these settings ensure compatibility with ES6
import/export */
            "esm": true,
        "compilerOptions": {
            "module": "CommonJS"
        }
    }
}
```

4️⃣ Version control

Ok we're nearly there. Let's get things set up to be tracked in version control and we can make a commit at this initial setup phase.

👉 Making sure you're in the project root folder, initialise the git repository:

```
git init -b main
```

👉 Git `add` and `commit` all the files you've created

```
git add -A && git commit -m "feat: Initial project setup"
```

💡 `git add -A` adds all files in the project folder regardless of which subfolder you run it in

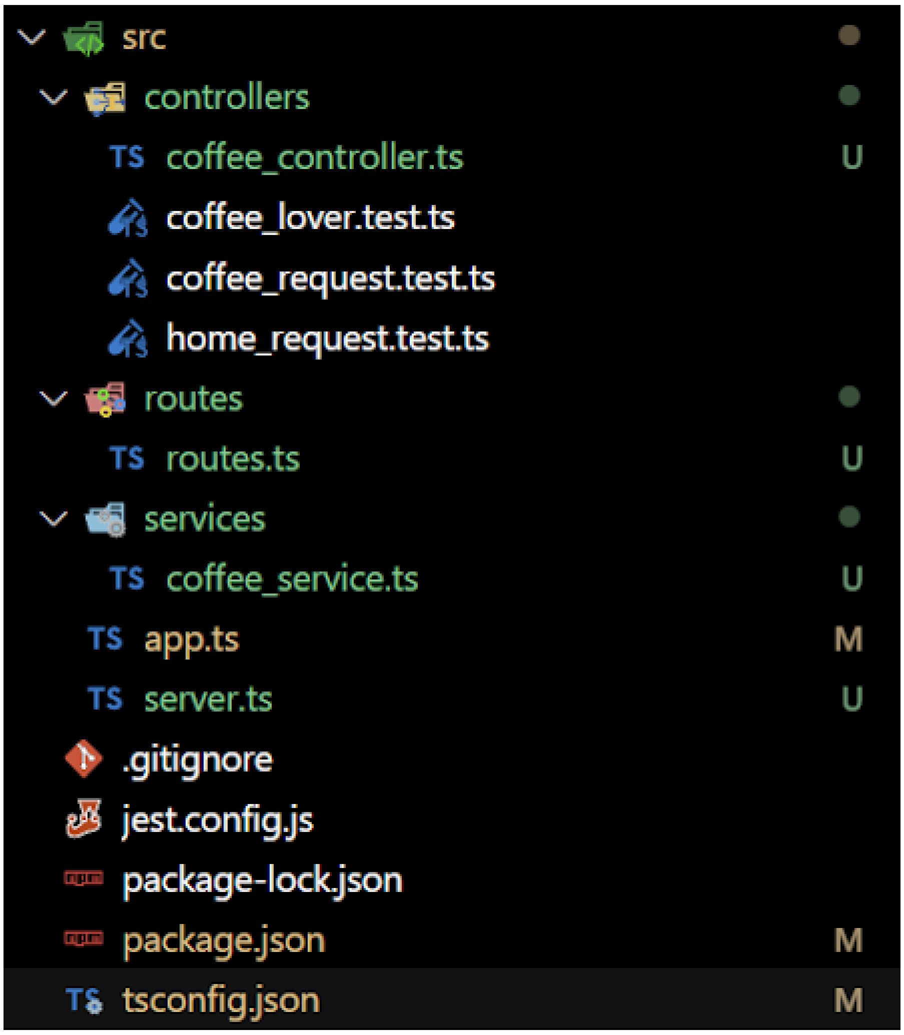
💡 That example commit message is using Conventional Commits - Conventional Commits - an industry-standard method of writing commit messages which allows for autogenerating patchnotes. We won't be doing that, and it's completely optional, but using it won't hurt and encourages good commit habits.

5️⃣ Create the Home Controller and the GET Endpoint for `/` to Give a String Response

Rather than spiking our web server, we're going to use Test-Driven Development.

📖 Prototyping new ideas and concepts in code is called *spiking*. A good guide to creating quality software is that when you finish spiking, you can use the concepts explored in the spike to re-engineer your solution from scratch with testing in mind.

Over the next few steps you will create a Drinks Web API that looks something like this, with all your project files in the appropriate folder and all your tests passing:



```
src
  controllers
    coffee_controller.ts       U
    coffee_lover.test.ts
    coffee_request.test.ts
    home_request.test.ts
  routes
    routes.ts                  U
  services
    coffee_service.ts          U
  app.ts                       M
  server.ts                    U
.gitignore
jest.config.js
package-lock.json
package.json                   M
tsconfig.json                  M
```

Write a test for the / API Endpoint

🔎 Let's write a test for the / API Endpoint. Look at the business requirement below to see what you need to test for:

As a user, when I do a GET request to the / endpoint, I will see a string message as a response which says `Welcome to the Drinks API!`

👉 Create a new directory in the route of your project called **src.** Within the **src** directory create a directory called **controllers**.

👉 Create a file called **home_request.test.ts** within the **src/controllers** directory

👉 Add the following code to the **home_request.test.ts** file:

```
import request from 'supertest';
import { app } from '../app';
// ⚠️ This should error as "../app" doesn't exist yet!

describe('Test home API endpoint request', () => {
  test('GET should return correct message', async () => {
    const res = await request(app).get('/');
    expect(res.statusCode).toEqual(200);
    expect(res.text).toEqual('Welcome to the Drinks API!');
  });
});
```

👉 Let's run this code and see what happens. Since it's a test file, Jest will pick it up when we run our test suites. In your terminal, run the tests:

```
npm test
```

👉 Run the test, you will see the test fail. In fact, if you pay attention to the error message, you'll see that Jest never even gets as far as our actual test, because it fails on the second `import` line. This is expected, as our test imports code that we haven't even written yet! Of course it doesn't work!

Let's start work on making the test pass.

Our test file is failing because we're importing code from **../app**. There's nothing special about that path, it's simply the name of the file we've chosen to define the start point of our API application. It's pretty common for JavaScript/TypeScript applications to name their entry point as either `app.ts` or `main.ts` or `index.ts`

💡 In Express applications, `app` is very common, so we'll stick with that.

👉 Create a file called **app.ts** within the **src** directory

👉 Put the following contents within that file

```
import express from 'express';

export const app = express();

app.get('/', (req, res) => res.send('Welcome to the Drinks
API!'));
```

⚠️ These few lines of code do a huge amount of work!

On line 3, we initialise an Express server in a variable which we are naming `app`. We `export` this variable so it is available throughout the rest of our application. (At the moment we're only using it in our test file, but as the app grows we will need access to this variable in a few other places.)

Take a moment to understand line 5. The server that is created by Express has some methods which can be used to setup endpoints, including `.get` and `.post`. Each of these "listens" for a specific HTTP verb on a specific path.

Line 5 tells the server to listen for a GET request on the path of **/** , and then provides a function which responds to that request. This function is known as a **handler** because it handles the request.

Conventionally, the first two parameters on each handler are named `req` and `res` - short for Request and Response. The Request contains all of the data sent to the server in the original request, including HTTP headers, query parameters, and potentially some JSON data. The server we are writing can then use that data to figure out what to send back via the `res` parameter, i.e. we craft a fresh Response depending on the contents of the Request.

Our handler sends a simple response. No matter what data is included in the request, it returns a string saying “Welcome to the Drinks API!”

👉 **Compare Line 5 to our original business requirement:**

❗ As a user, when I do a GET request to the `/` endpoint, I will see a string message as a response which says `Welcome to the Drinks API!`

🥳 It sure seems like line 5 encapsulates that logic very neatly!

👉 Now try to run your tests again. They should now pass 🙌

👉 Make sure your test is ACTUALLY testing the right thing by breaking it! Change the expected test in some way:

```
expect(res.text).toEqual('This should fail!');
```

🥳 It does fail! This helps build confidence that we haven’t accidentally written a test that always passes, which obviously isn’t very useful.

👉 Finally let's get our work committed to Git:

```
git add -A && git commit -m "feat: Implement GET request for home endpoint and test"
```

6️⃣Create the Coffee Controller and the GET Endpoint for `/coffeelover`

Write a test for the `/coffeelover` API Endpoint

Now its over to you! Using what you've learnt so far, write a test for the `/coffeelover` API Endpoint. Look at the business requirement below to see what you need to test for.

As a user, when I do a GET request to the `/coffeelover` **endpoint, I will see a string message as a response which says** `I like coffee!`

👉 Using a similar approach to the previous section, have a go at doing this yourself!

👉 Write your test first and then write the code to make your test pass.

👉 Don't forget to commit your changes to version control before moving to step 7.

One possible solution can be viewed in the **Solutions** section of the lab towards the end of this document.

7️⃣Create a GET Endpoint to serve JSON data back as a Response to a Request with a Request Parameter (Request Param)

At this stage we should have two tests, which test two endpoints that can be invoked by sending GET requests to the following routes '*/*' and '*/coffeelover*'

Both of these endpoints always send back a simple string, no matter what.

Next we're going to create a new endpoint which accepts a **request parameter**. This allows the server to be a little more advanced in how it responds.

The path for our new endpoint will be **/coffee** and it will accept a request parameter named **coffeeName**. In response it will return a JSON object that looks like this:

```
// ℹ️ this isn't valid JSON because it includes comments 😮

{
  drinkType: 'Coffee',     // always "Coffee"
  name: 'Latte',           // return whatever name was passed
in
}
```

Our user story is defined as the following:

As a user, when I do a GET request to the `/coffee` endpoint, I will see the JSON response which includes two properties: the drinkType set to "Coffee" and name set to the provided name

- If a value is not provided for the `coffeeName` parameter as part of the Request Params, the `name` of the coffee in the JSON response should be `Latte` by default.

- If a value is provided for the `coffeeName` parameter as part of the Request Params, the `name` of the coffee is the `name` given by the user. For example, if the value of the `coffeeName` Request Param is `cappuccino` and this was provided on the `GET` request, the JSON response should have `cappuccino`.

👉 This is TDD, so let's begin by creating a test.

👉 Create a file named **coffee_controller.test.js** in the **src/controllers** folder, to test the **/coffee** endpoint. (We don't yet have a full coffee controller, but we will soon!)

👉Add the following test:

```
import request from 'supertest';
import { app } from '../app';

test('GET /coffee should return correct object', async () => {
  const res = await request(app)
  .get('/coffee')
  .query({ coffeeName: 'Latte' });

  expect(res.statusCode).toEqual(200);
  expect(res.body).toEqual({
    drinkType: 'Coffee',
    name: 'Latte',
  });
});
```

👉 Run your tests and make sure this new test is failing.

In the test code above we are expecting a response to a new request, this time sending a GET to the /coffee endpoint. Notice that we also pass along a query parameter for the coffee being requested.

The `.query` syntax to pass along the parameter is specific to `supertest`. In the real world, perhaps this parameter would be supplied via a call to `fetch` in the browser. Any software that can make HTTP requests can call this endpoint and supply a query parameter.

The assertion in our new test checks for that the response body has a JSON object with the correct properties set.

Now let's get this test to pass.

👉 Add the following line (around line 4 after you've declared the `app` variable) to your **app.ts**

```
app.use(express.json());
```

This line tells Express to enable its built-in JSON parser, which we will need if we start POSTing JSON to it later on.

You can think of this `app` variable as like building a set of instructions for Express: "If anybody contacts you with JSON, that's alright! You can use your built-in method to parse it". In future we'll see how those instructions can become more flexible and powerful… but for now we can go a long way by defining simple **handlers** for different requests.

Let's create the new endpoint. Add the following to your **app.ts**:

```
app.get('/coffee', (req, res) => res.json({
    drinkType: 'Coffee',
    name: 'Latte',
}));
```

This time we respond using `res.json`, which as you might expect returns a JSON object in the format of our required response. Currently its hard coded to return the name of **Latte** - so we haven’t yet met all the business requirements, but we are meeting the requirements of our initial test.

👉 Run your tests again and they should now pass.

👉 Now your tests are passing, commit your code to Git.

8️⃣Refactoring our API to MVC and introduce a separation of concerns

In step 7 you created an API endpoint to handle GET requests for /coffee and return JSON.

However the structure of our code can be improved. Currently we've put all our code within the **app.ts** file. Essentially we haven't *separated our concerns* - everything is included in a single file.

Firstly, let's separate our routes (the paths that our server is listening to) from the controllers (the functions which handle the request).

As applications grow in size, separation of concerns becomes even more important. In general, each element should have one, simple job:

Routes - The paths that our server cares about, e.g. "/" or "/home"

Controllers - Maps a route to the business logic. Controllers understand how to get the right data from a Request and what to put in a Response. But they don't know anything about the actual application logic! That is the job of…

Services - The actual business logic. These don't know anything about routes or requests - they understand things like *"I get the correct flavour of coffee"* or *"I retrieve the product by an id"*. The service shouldn't know about where the data is actually stored, that is the job of…

Models - The underlying data layer. The services don't know if data is in a database or a file or stored in memory or retrieved from some other system entirely. All of that is handled by Models, which do things like *"get a product from the database by id"* or *"update the name of product with id [x]"*.

👀 Notice how each of these is SUPER simple. This allows for any complexity to be kept localised - e.g. if there's a particularly complex HTTP Request to decipher, then that is the job of the Controller to figure out, then it asks the Service for the relevant data.

This current application won't require so many layers! But we can still do a better job of separating our concerns than we are at the moment.

👉 Create two new directories in the **src** directory called **routes** and **services**. So now you should have the following directories **src/routes**, **src/services** and **src/controllers**

👉 In the **controllers** directory create a new file called **coffee_controller.ts** and put the following contents within it:

```
import Express from 'express';
import * as coffeeService from '../services/coffee_service';

export const getCoffee = async (req: Express.Request, res: Express.Response) => {
     const { coffeeName } = req.query;
     const coffee = coffeeService.getCoffee(coffeeName as string);
     res.json(coffee).status(200);
};
```

👀 Notice we import the **coffeeService**. Again, this is something we are planning to create shortly.

👀 Also notice we are importing `Express` so we can refer to the types - Express defines the types it is expecting for `req` and `res`, so that allows us to annotate our function and get the benefits of strong-typing in our functions. Typing `req.` inside the function now gives us some lovely autocomplete 🥳

👉 Rather than importing the whole `Express` object to refer to the types, we can *destructure* our import to grab the types directly:

```
import { Request, Response } from 'express';
// ❗ altered import
import * as coffeeService from '../services/coffee_service';

export const getCoffee = async (req: Request, res: Response)
=> {
     const coffeeName  = req.query.coffeeName;
     const coffee = coffeeService.getCoffee(coffeeName as
string);
     res.json(coffee).status(200);
};
```

This makes the code a little neater, but the result is identical, so go with whichever you prefer.

👀 Our function called **getCoffee** accepts a request and response. On line 5 it reads the request and accesses the query parameters to obtain the **coffeeName**. Then the coffeeName is passed on to the coffee service which will (soon) produce the JSON object.

On line 7 that JSON object is then sent back in the response, along with setting the status code to 200 (success).

Look at line 6. It's a little suspicious that we're enforcing to TypeScript that `coffeeName` should be considered always as a `string`. After all, it could certainly be `undefined` - that's the whole point of having a default parameter! And query parameters can also be `string[]` or even all kinds of other possible types!

However, since we know we're being very restrictive with how we call our endpoints we could safely assume it's a `string` in this case… but…

Let’s be good to our future selves and fix that!

The developers who wrote the `@types/express` library foresaw that developers would want to specify which types are expected in query parameters, so they made `Request` generic.

💡 In fact, `Request` takes *multiple* generic parameters which allows developer to specify the exact types which are expected for many different parts of the request.

However, for our purposes, we only care about the type of the query parameters, which happens to be the **fourth** generic parameter to the `Request` type:

```
Request<T, U, V, QueryParameterTypes>

// let's not worry about what T, U and V are - think of them as parameters that don't matter for our current purposes... because that's what they are!
```

👉 Change the type annotation of `req` in **coffee_controller.ts** from just plain `Request` to `Request<object, object, object, { coffeeName: string | undefined }>`

This keeps the first three parameters as they were - totally unconstrained, they can be any `object`. And it specifies the precise shape of our `queryParameters` : they will be an object with one property, named `coffeeName` which could be either `string` or `undefined`.

👉 Now we can remove the `as string` from the call to `getCoffee` 🥳 This means that TypeScript can enforce that we only call the coffee controller with valid parameters!

Your **coffee_controller.ts** file should now look something like this:

```
import { Request, Response } from 'express';
import * as coffeeService from '../services/coffee_service';
```

```
export const getCoffee = async (
     req: Request<object, object, object, { coffeeName: string
| undefined }>,
     res: Response
) => {
     const coffeeName = req.query.coffeeName;
     const coffee = coffeeService.getCoffee(coffeeName);
     res.json(coffee).status(200);
};
```

Now let's create that coffee service.

👉 In the **services** directory create a new file called **coffee_service.ts** and put the following contents within it:

```
export const getCoffee = (name = 'Latte') => {
     return {
          drinkType: 'Coffee',
          name,
     };
};
```

Recall that **the job of a service is to encapsulate the business logic**, and to ask the model layer for any data required to do that. In our case, we have no persistent data storage, and therefore no model layer. So the coffee service has a simple job: encode any logic to do with coffee.

The business logic in this case is also simple. At the moment, it requires just one function called **getCoffee** which returns a JSON object. Notice we use a default parameter for **name** to ensure we meet the requirements of the user story.

Any other logic to do with 'coffee' in this application as it grows would also be encapsulated in this service. If we later added persistent storage then that would be placed in a new `model` layer, and this service would then have the additional job of communicating with that layer too.

The key point here is that it’s **not the controllers job to handle business logic** like:

- “default coffee name is Latte”
- “the JSON object has two properties, `drinkType` and `name`"
- “the `name` property on the response is set to the input coffee name”

These three points are all super simple, but they are still business logic, so we put them in a service layer. This separation of concerns helps our application to remain simple even when growing to a massive size, as **each part of it locally has a small, easily-comprehended job** to do.

Finally let's set up a dedicated file for hooking up our routes to the relevant controllers.

👉 In the **routes** directory create a new file called **routes.ts** and put the following contents within it:

```
import express from 'express';
import * as coffeeController from
'../controllers/coffee_controller';

export const router = express.Router();

router.get('/coffee', coffeeController.getCoffee);
```

This file sets up a route mapping: the path **/coffee** will call the **getCoffee** function of the **coffeeController**

The final step is to bind this router to our Express application.

👉 Update your **app.ts** and include a new import to import the routes. Add the following line to the list of imports:

```
import { router } from './routes/routes';
```

👉 We are leaving the home endpoint and the `coffeelover` endpoint the same, as we haven't created controllers for those.

👉 Replace the **/coffee** request mapping in the **app.ts** with our new route mapping. Replace the following code block:

```
app.get('/coffee', (req, res) =>
     res.json({
          drinkType: 'Coffee',
          name: 'Latte',
     })
);
```

With:

```
app.use('/', router);
```

Your **app.ts** should now look like this:

```
import express from 'express';
import { router } from './routes/routes';

export const app = express();

app.use(express.json());

app.get('/', (req, res) => res.send('Welcome to the Drinks
API!'));
app.get('/coffeelover', (req, res) => res.send('I like
coffee!'));

app.use('/', router);
```

👉 Run your unit tests - they should all still be passing.

🎉 You've successfully refactored your code, separated concerns, and ensured your unit tests continue to pass! 🥳

👉 Git commit your code to version control

Design Considerations

For such a small app, it might seem like overkill to set up all these separate **routes** and **service** and **controller** layers. But this allows the app to grow in a scalable manner.

👉 If new 'coffee-based' functionality is required, it can be added to the **coffee_service** and **coffee_controller**. For example, we might add an endpoint to "add new flavour of coffee" which would then require adding to those two places, and then hooking up the route as before.

👉 If new "non-coffee-based" functionality is required, we can add new services and controllers. For example, if we added "tea" and "juice" and other forms of drinks, we could add corresponding services, controllers and routes.

👉 We could also move our existing `/` and `coffeelover` endpoints to their own services and controllers to use the same pattern everywhere in our codebase.

All of this together helps our app to remain simple even as it scales up.

It's easy to write a new controller - it just has to parse data from the Request and then ask the relevant service to figure out what to send back. It's easy to write new service function - it just takes a given input (e.g. a name of a coffee, or the id of a product, or a bunch of data sent from the client) and figures out what to return.

Each layer has a simple job, making it easy both to write and to understand.

Running as a Full Server

At the moment our application only executes via tests. It's great to know that each of our endpoints works, but the final step is to make the server actually run as a fully-fledged server.

👉 Create a new file within the **src** directory called **server.ts** and place the following contents within that file:

```
import { app } from './app';

app.listen(3000, () => {
     console.log('Express server started on port 3000');
});
```

Here we simply import the app server we've been building and we start it listening on port 3000. (This is a commonly used port for development, but we could choose any number we like here.)

The final change we can make is to tell node how to start our server. Just as we did with the `npm test` command, we'll use the **scripts** section of `package.json` and introduce a **start** script.

👉 Update the **scripts** section of the **package.json** so that it has the following content:

```
"scripts": {
  "test": "jest",
  "start" : "nodemon src/server"
},
```

Notice we've introduced a new command called **start**. Now if we run `npm start` it will automatically invoke whatever command is defined. In the case above that is `nodemon src/server`.

👉 Start your server by running start

```
npm start
```

You should now be able to navigate to http://localhost:3000/ and see your wonderful API. 🎉

Let’s try some of the other end points 🙌

👉 Try navigating to http://localhost:3000/coffeelover/ and http://localhost:3000/coffee/?coffeeName=mocha too

👉 You could also call those endpoints using Postman!

👉 What happens if you try to GET from a non-existent endpoint? 🤔

👉 Git commit your code to version control

9️⃣Write extra tests for the coffee controller and push to GitHub

👉 Write more tests for making requests to the coffee endpoint.

- Currently we’re only testing with a query parameter which is hardcoded to the default parameter, so we are missing testing some codepaths.
- You could try a test with a **different coffee name**
- You could try a test where the **coffee name parameter is not specified**

The Solution can be viewed in the **Solutions** section of the lab towards the end of this document.

👉 Create a new **public blank** (don't initialise it with a README or license etc) repository on GitHub to contain your API, lets call it **drinks-service-api**

👉 Follow the instructions to **push an existing repository from the command line**

🔟 Extension exercises

👉 Using what you have learnt see if you can refactor the **/coffeeLover** endpoint to make use of the new controller.

👉 Using what you have learnt, introduce a new **/tea** endpoint

⭐ 🎉 That’s it! Woohoo! Well done! Don’t forget to submit the link to your Github repo containing your API!

You can view a demo solution for today’s lab here:

🎈 Solutions

💠 Solution for step 6: The `coffeelover` endpoint

👉 You should have created a file called something similar to **coffee_controller.test.ts** located in the **controllers** directory

👉 It should have content similar to the following:

```
// coffee_controller.test.ts

import request from 'supertest';
import { app } from '../app';

describe('Test coffee API endpoint request', () => {
  test('GET /coffeelover should return correct message', async
() => {
    const res = await request(app).get('/coffeelover');
    expect(res.statusCode).toEqual(200);
    expect(res.text).toEqual('I like coffee!');
  });
});
```

👉 The **app.ts** file should be updated to include the following:

```
app.get('/coffeelover', (req, res) => res.send('I like
coffee!'));
```

💠 Solution for step 9 extra tests for the coffee endpoint

You should add further tests in the **coffee_controller.test.ts** file. Similar to the example below:

```
test('GET /coffee with no param should return correct object',
async () => {
  const res = await request(app)
  .get('/coffee')

  expect(res.statusCode).toEqual(200);
  expect(res.body).toEqual({
    drinkType: 'Coffee',
    name: 'Latte',
 });
});

test('GET /coffee with different param should return correct
object', async () => {
  const res = await request(app)
  .get('/coffee')
  .query({ coffeeName: 'Macchiato' });

  expect(res.statusCode).toEqual(200);
  expect(res.body).toEqual({
    drinkType: 'Coffee',
    name: 'Macchiato',
  });
});
```