Tech Returners

🧪👾 Welcome to TS environment set-up

In this lab, you'll learn how to turn an empty folder into a TypeScript environment with node and Jest! 🥳💻

🛠️ What do I need for this Lab?

You will need:

- Visual Studio Code
- ts-node
- ts-jest
- npx

🤔Running Jest with Typescript?

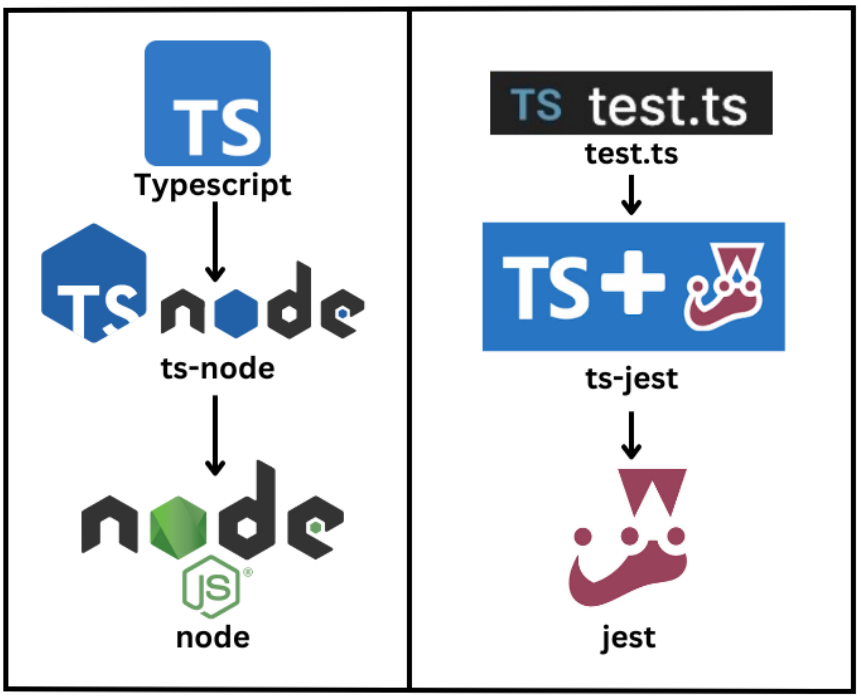
We've seen how our TypeScript code relies on a "pipeline" to run. Because `node` doesn't speak TypeScript, we have to set up some process to convert it to JavaScript and then pass it to node.

In particular, we've seen how to use `ts-node` to convert our TypeScript into JavaScript that `node` can understand.

But what about our tests? We might hope that hooking up `ts-node` will magically make our tests work too. Sadly, that isn't the case!😔

Think about how our test code usually works when we write JavaScript. Jest loads a file called **something.test.js** and runs it. That's all that happens! Our nice TS pipeline where code passes through ts-node just doesn't exist for test code... yet 🤓

So, let's make it! The missing piece will be `ts-jest` which will do a similar job to `ts-node`.



We can think of our code + tests as needing two separate pipelines - one for application code, and one for tests.

1️⃣ Create a new project

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

👉 Firstly create a new directory to house your code. Let's call the directory the **ts-project-with-jest:**

```
mkdir ts-project-with-jest
```

👉 Using the terminal, navigate to your **ts-project-with-jest** directory:

```
cd ts-project-with-jest
```

👉 Whilst in the **ts-project-with-jest** directory, initialise the project by running:

```
npm init -y
```

💡 *The `-y` flag skips questions that we aren't concerned with currently.*

👉 Install and add TypeScript as a dev dependency by running :

```
npm i -D typescript
```

You should now have both a **package.json** and a **package-lock.json**

📖 **Package.json -** Contains descriptive and functional metadata about a project, such as a name, version, and dependencies. This config file gets automatically altered if we `npm install` a new package, but we can also manually edit it to change dependencies, versions, or other config settings.

Package-lock.json - Records the exact version of every installed dependency. We don't normally work with this ourselves - this is *auto-generated* by **npm** from the package.json.

2️⃣ Setup tsconfig file

We've installed TypeScript, but we haven't told it to actually *do* anything. One important file which tells TypeScript how to behave in a project is called **tsconfig.json**

👉 To setup the tsconfig file run the following command:

```
npx tsc --init
```

💡 *Remember: `npx` executes a package and `tsc` is the typescript compiler*

This sets up our folder as a TypeScript project, by adding a **tsconfig.json** with some default configuration.

The tsconfig is a complicated file with many, many options. Luckily, we don't have to be experts in `tsconfig`. We just need to be aware it exists and understand some of the important options!

💡 **Take some time to notice these options in your tsconfig.json:**

- `target` - what version of JS we want to output. Depending on where our JS will be running, we might want to restrict this. For example, if our program is going to run on an outdated browser or an old version of node, we can specify which version of JavaScript we want to appear here. Or we can go for the very latest JS, like **`ES2023`**
- `module` - Should our JavaScript use ES modules or CommonJS modules? We can choose `ES2015` for ES modules.
- `esModuleInterop : true` - This is helpful. It emits additional JS to ease support for importing CommonJS modules.
- `strict: true` - Checks behaviour to ensure program correctness
- `rootDir` and `outDir` - These specify where our output JS should go

Some very important tsconfig options are **include** and **exclude**. You might have guessed that these determine which files we want TypeScript to include or exclude in the compilation. Make sure you have them set in your tsconfig:

👉 `include` - the src folder

```
“include”: [“./src/**/*”]
```

👉 `exclude` - anything we don’t want directly included, such as node_modules

```
“exclude”: [“node_modules”]
```

Let’s update the `rootDir` and `outDir` to add a path for the outputs

👉 `rootDir`

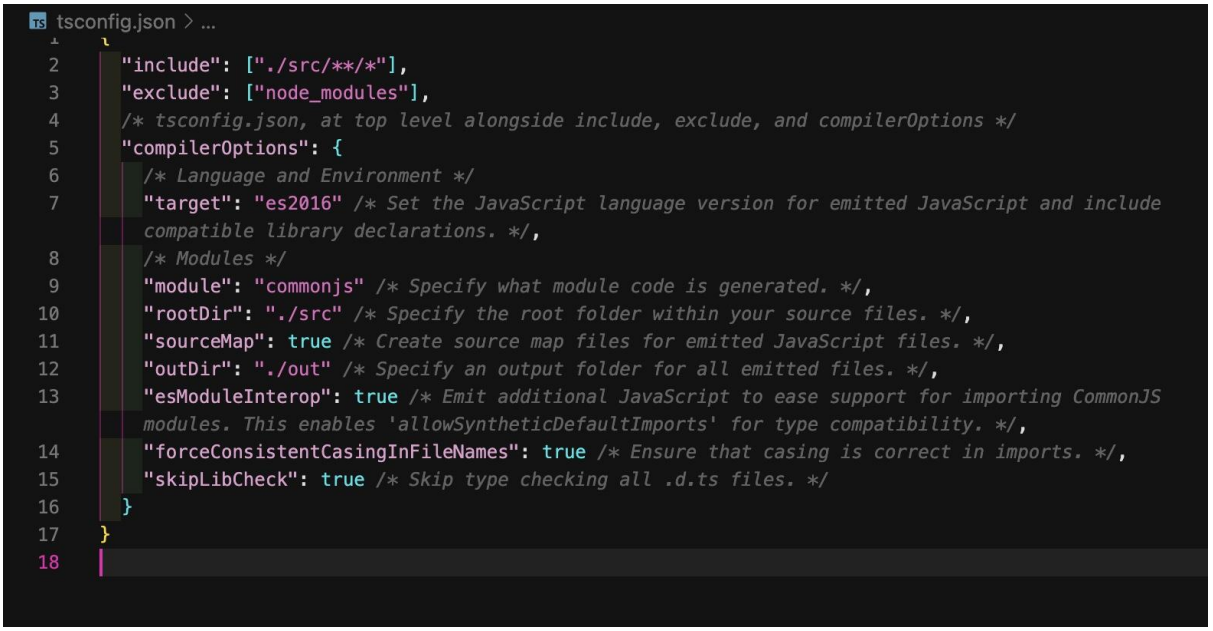
```
“rootDir”: "./src"
```

👉 `outDir`

```
“outDir”: "./out"
```

The tsconfig.json should look something like this (along with commented out properties):

tsconfig.json > ...



```jsonc
{
  "include": ["./src/**/*"],
  "exclude": ["node_modules"],
  /* tsconfig.json, at top level alongside include, exclude, and compilerOptions */
  "compilerOptions": {
    /* Language and Environment */
    "target": "es2016" /* Set the JavaScript language version for emitted JavaScript and include compatible library declarations. */,
    /* Modules */
    "module": "commonjs" /* Specify what module code is generated. */,
    "rootDir": "./src" /* Specify the root folder within your source files. */,
    "sourceMap": true /* Create source map files for emitted JavaScript files. */,
    "outDir": "./out" /* Specify an output folder for all emitted files. */,
    "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. */
  }
}
```

3️⃣ Application Pipeline: ts-node

Time to get our application pipeline up and running. We're going to use ts-node together with nodemon to run our program more conveniently!

👉Install ts-node and nodemon, and the types package for node itself:

```
"npm i -D ts-node nodemon @types/node"
```

👉 Add some `ts-node` config to our **`tsconfig.json.`**

This should be at the top level of **`tsconfig.json`** with `include` and `exclude` outside the compiler options.

```
"ts-node": {
            "esm": true,
            "compilerOptions": {
                  "module": "CommonJS"
            }
      }
```

Your tsconfig.json should now look something like this:

📖 This will let us use ES modules but it'll still output CommonJS to node, which makes life a bit easier with Jest, as Jest struggles with ES modules even in 2023 😔

Now let's try it out!

👉 Create a folder **src**

```
mkdir src
```

👉 Using the terminal, navigate to your **src** directory

```
cd src
```

👉Create a file **calc.ts**

On Mac/Linux you can use this command, or you can create a file in VS Code in the normal way you would

```
touch calc.ts
```

👉Create a file **index.ts**

```
touch index.ts
```

It's time to write some code. We could write an entire application, but for demonstration purposes let's keep it simple!

👉Add the following code to **calc.ts**:

```
export function add(x: number, y: number) : number {
return x + y;
}
```

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

```
import { add } from './calc';

console.log(add(2,2));
```

👉We can now add the following to our **scripts** inside our **package.json** in order to run our code.

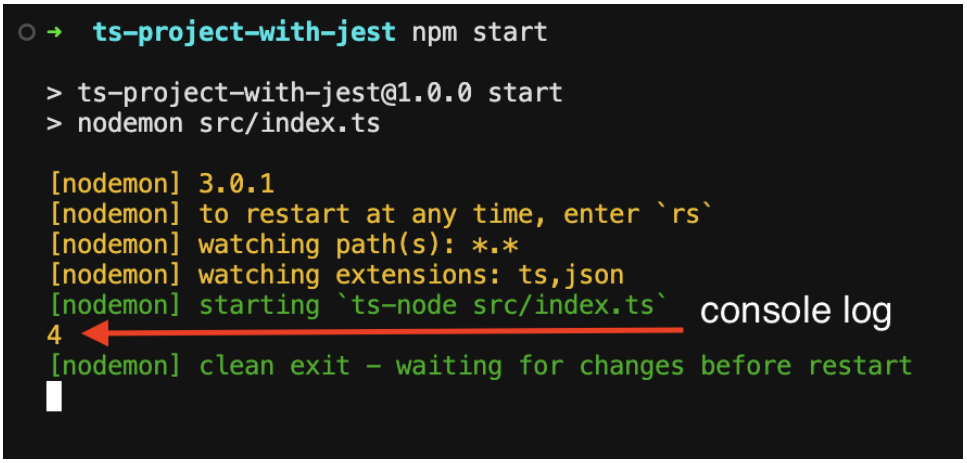
```
"start": "nodemon src/index.ts",
```

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "nodemon src/index.ts"
},
```

👉Let's try running our code, and we should now see our *console.log* in the terminal 🤞:

```
npm start
```

You should now hopefully see the following!



4️⃣ Test Pipeline: ts-jest

Of course, we'd never write a whole application without writing tests. Right?!

First we need to start by installing jest and something to translate our TypeScript into the JavaScript which Jest demands. this Just like with ts-node, we have options here, but we're going to go with ts-jest

👉To install, we need to run the following command. Notice the similarity to what we did to get our application pipeline working!

```
npm i -D jest ts-jest @types/jest
```

👉We also need to initialise a jest config too:

```
npx ts-jest config:init
```

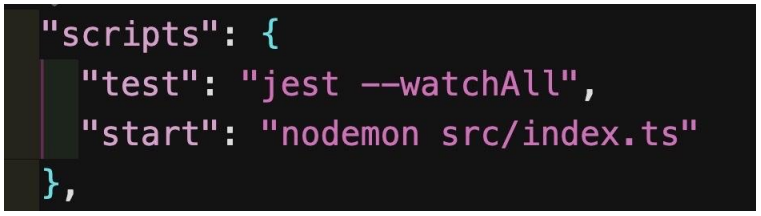
You should now see a **jest.config.js** file appear 🪄 - magic!

📖 Our jest.config.js generated by ts-jest includes a preset for jest to use the preprocessor ts-jest

In other words, before using any file, jest will run it through ts-jest, which is exactly what we want.

👉We now want to update our **test script** to run jest

```
    "test": "jest --watchAll"
```



💡 *The* --watchAll *flag will mean our tests will automatically run every time there is a change to a file*

Now jest is all set up and ready to roll, let's get to work on our tests!🧪

👉Create a new folder in your route called **tests**

```
mkdir tests
```

👉cd into your **tests** directory

```
cd tests
```

👉Create a new file named **index.test.ts**

```
touch index.test.ts
```

We can then open this file up and start working on writing our tests!

👉Add the following test case in your **index.test.ts file**

```
import { add } from "../src/calc";


describe("test add function", () => {
  it("should return 15 for add(10,5)", () => {
    expect(add(10, 5)).toBe(15);
  });it("should return 5 for add(2,3)", () => {
    expect(add(2, 3)).toBe(5);
  });
});
```

Time for the big moment! 🥁 Let's try running our test.

👉Run your test using the following command:

```
npm test
```

5️⃣ Launch settings

Yay! We have tests passing! 🎉 But what if we have bugs? How would we go about debugging these?

Sometimes our bugs are in our application code, sometimes they're in the tests.

There are numerous tools which help us track down problems. We're going to look at two: running tests in isolation, and the built in VS Code debugger.

Running Tests in Isolation

Sometimes we want to just run a single test, rather than every test in our project. One very convenient way to do this is the 🪲`Jest Runner` VS Code extension. This extension gives links next to each test or group of tests, so we can run them easily with a single click.

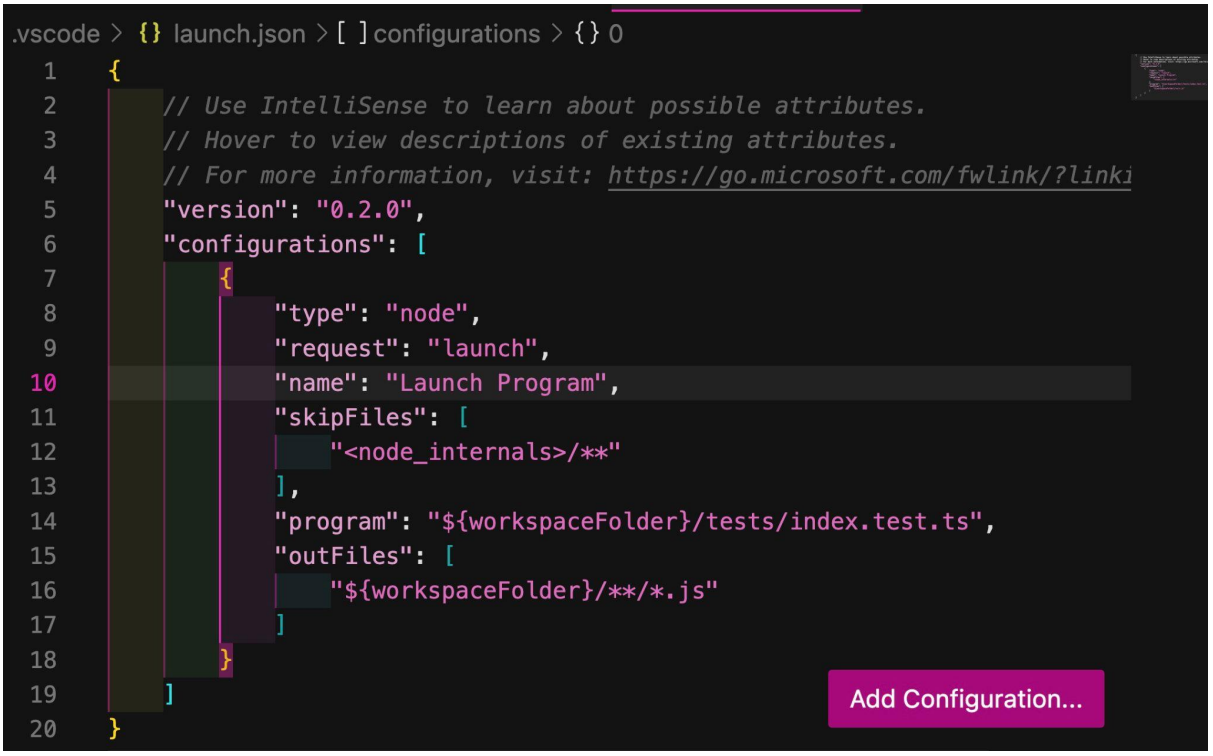
Debugging

👉Add sourceMap in **tsconfig.json compilerOptions** - this enables a source map to be generated so the debugger can match up the running code with the written code (*remember, the type of Javascript that is given out by running ts-node is not the same as the TypeScript we see in our code windows - sourceMap will bridge this gap for us!*)

```
"sourceMap": true
```

👉Open "Run and Debug" config in VS Code (`Ctrl-Shift-D` or `Cmd-Shift-D`) and select `Create a launch.json`

This creates a **.vscode/launch.json** - a file which tells VS Code how to run your project. It should look something like this:



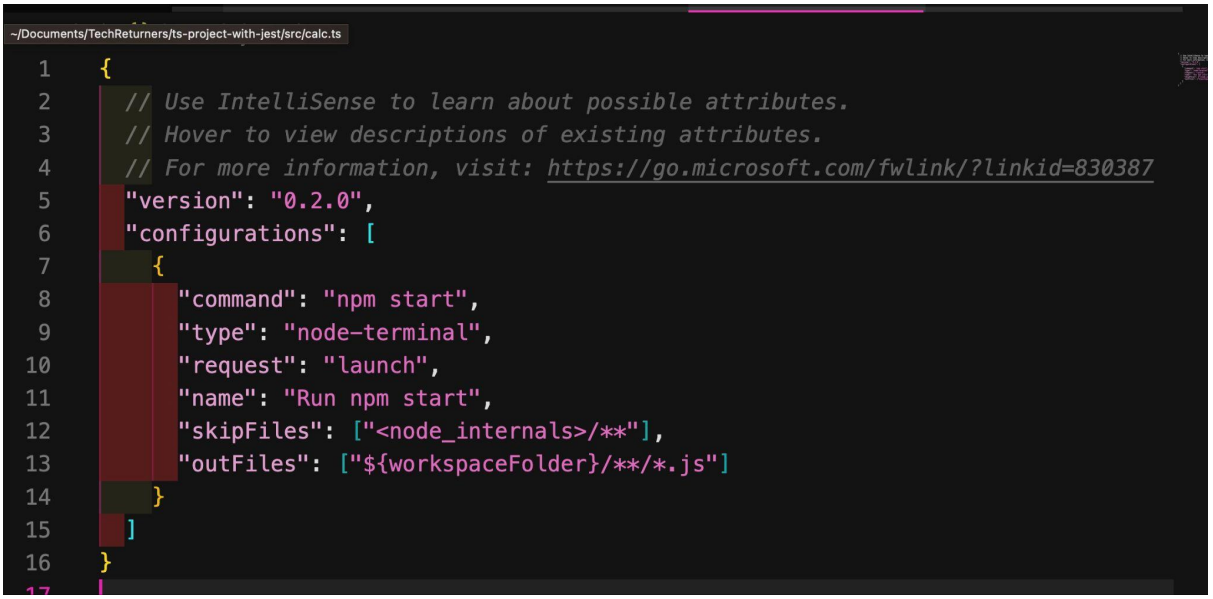
⚠️The above would work if we were running `js` directly in node, but we have an intermediate step of `ts-node`. Instead we can use a different command!

👉Update the **launch.json** configurations (‘`command`’, ‘`name`’ and ‘`type`’) to be the following:

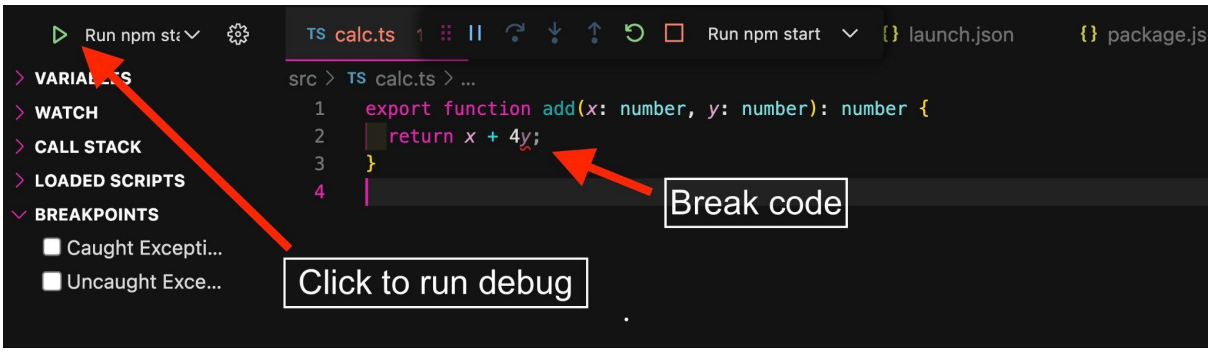
```
"command": "npm start",

"name": "Run npm start",

"type": "node-terminal"
```

💡 *You may also be prompted to remove the ‘program’ property from the launch.json - feel free to do so!*

Your **launch.json** should now look like this:



Now we can set a breakpoint in our calc function and use “Run & Debug” to be able to see what’s happening at the time.



Depending on our build tools this can get a bit complex, but we can usually find a `launch.json` that allows this - however, it can usually be an easier option to simply `console.log` debug.

⚠️ A common error you might see:

‘Cannot launch program because corresponding JavaScript cannot be found’

You've likely not set `"sourceMap": true` in your **`tsconfig.json`** or `outFiles` in your **`launch.json`** and the VS Code Node.js debugger can't map your TypeScript source code to the running JavaScript. Turn on source maps and rebuild your project