b First steps with TypeScript

After the brief introduction to the main principles of TypeScript, we are now ready to start our journey towards becoming FullStack TypeScript developers. Rather than giving you a thorough introduction to all aspects of TypeScript, in this part we will focus on the most common issues that arise when developing express backends or React frontends with TypeScript. In addition to language features we will also have a strong emphasis in tooling.

Setting things up

Install TypeScript support to your editor of choice. Visual Studio Code works natively with TypeScript.

As mentioned earlier, TypeScript code is not executable by itself but it has to be first compiled into executable JavaScript. When TypeScript is compiled into JavaScript, the code becomes subject for type erasure. This means that type annotations, interfaces, type aliases, and other type system constructs are removed from the code and the result is pure ready-to-run JavaScript.

In a production environment the need for compilation often means that you have to setup a "build step". During the build step all TypeScript code is compiled into JavaScript in a separate folder, and the production environment then runs the code from that folder. In a development environment it is often more handy to make use of real-time compilation and auto-reloading in order to be able to see the resulting changes faster.

Let's start writing our first TypeScript-app. To keep things simple, let's start by using the npm package ts-node. It compiles and executes the specified TypeScript file immediately, so that there is no need for a separate compilation step.

You can install both *ts-node* and the official *typescript* package globally by running

```
npm install -g ts-node typescript
```

If you can't or don't want to install global packages, you can create an npm project which has the required dependencies and run your scripts in it. We will also take this approach.

As we remember from part 3 an npm project is set by running the command *npm init* in an empty directory. Then we can install the dependencies by running

```
npm install --save-dev ts-node typescript
```

and set up *scripts* within the package.json:

```
{
  // ..
  "scripts": {
    "ts-node": "ts-node"
  },
  // ..
}
```

Now within this directory you can use *ts-node* by running *npm run ts-node*. Note that if you are using ts-node through package.json, all command line arguments for the script need to be prefixed with --. So if you want to run file.ts with *ts-node*, the whole command is:

```
npm run ts-node -- file.ts
```

It is worth mentioning that TypeScript also provides an online playground, where you can quickly try out TypeScript code and instantly see the resulting JavaScript and possible compilation errors. You can access TypeScript's official playground here.

NB: The playground might contain different tsconfig rules (which will be introduced later) than your local environment, which is why you might see different warnings there compared to your local environment. The playground's tsconfig is modifiable through the config dropdown menu.

A note about the coding style

JavaScript on itself is quite relaxed language, and things can often be done in multiple different ways. For example we have named vs anonymous functions, using const and let or var and the use of *semicolons*. This part of the course differs from the rest by using semicolons. It is not a TypeScript specific pattern but a general coding style decision taken when creating any kind of JavaScript project. Whether to use them or not is usually in the hands of the programmer, but since it is expected to adapt ones coding habits to the existing codebase, in the exercises of this part it is expected to use semicolons and to adjust to the coding style of the part. This part has some other coding style differences compared to the rest of the course as well, e.g. in the directory naming.

Let's start by creating a simple Multiplier. It looks exactly as it would in JavaScript.

```
const multiplicator = (a, b, printText) => {
  console.log(printText,  a * b);
}

multiplicator(2, 4, 'Multiplied numbers 2 and 4, the result is:');
```

As you can see, this is still ordinary basic JavaScript with no additional TS features. It compiles and runs nicely with *npm run ts-node -- multiplier.ts*, as it would with Node. But what happens if we end up passing wrong *types* of arguments to the multiplicator function?

Let's try it out!

```
const multiplicator = (a, b, printText) => {
  console.log(printText,  a * b);
}

multiplicator('how about a string?', 4, 'Multiplied a string and 4, the result is:');
```

Now when we run the code, the output is: *Multiplied a string and 4, the result is: NaN*.

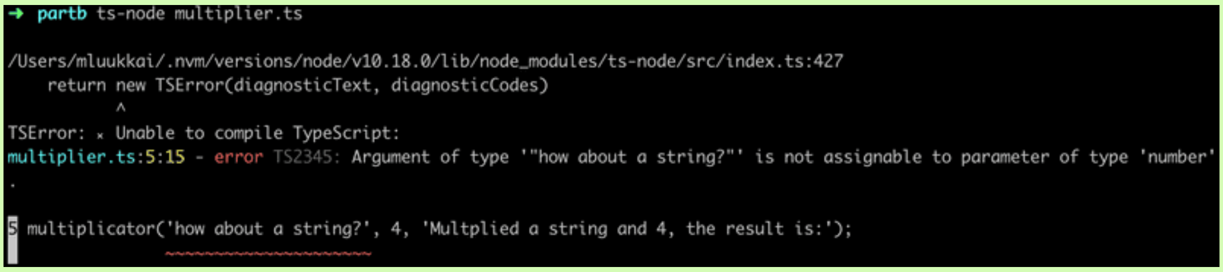
Wouldn't it be nice if the language itself could prevent us from ending up in situations like this? This is where we see the first benefits of TypeScript. Let's add types to the parameters and see where it takes us.

TypeScript natively supports multiple types including *number*, *string* and *Array*. See the comprehensive list here. More complex custom types can also be created.

The first two parameters of our function are of the type number and the last is a string:

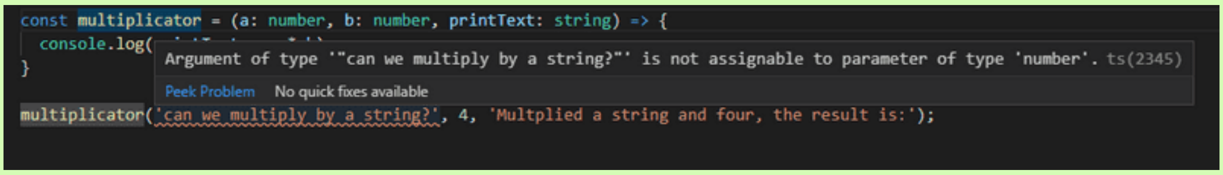
```
const multiplicator = (a: number, b: number, printText: string) => {
  console.log(printText,  a * b);
}

multiplicator('how about a string?', 4, 'Multiplied a string and 4, the result is:');
```

Now the code is no longer valid JavaScript, but in fact TypeScript. When we try to run the code, we notice that it does not compile:



```
➜ partb ts-node multiplier.ts

/Users/mluukkai/.nvm/versions/node/v10.18.0/lib/node_modules/ts-node/src/index.ts:427
    return new TSError(diagnosticText, diagnosticCodes)
           ^
TSError: ⨯ Unable to compile TypeScript:
multiplier.ts:5:15 - error TS2345: Argument of type '"how about a string?"' is not assignable to parameter of type 'number'
.

5 multiplicator('how about a string?', 4, 'Multplied a string and 4, the result is:');
```

One of the best things in TypeScript's editor support is that you don't necessarily need to even run the code to see the issues. The VSCode plugin is so efficient, that it informs you immediately when you are trying to use an incorrect type:



```
const multiplicator = (a: number, b: number, printText: string) => {
  console.log(
}
multiplicator('can we multiply by a string?', 4, 'Multplied a string and four, the result is:');
```

```
Argument of type '"can we multiply by a string?"' is not assignable to parameter of type 'number'. ts(2345)
```

Creating your first own types

Let's expand our multiplicator into a bit more versatile calculator that also supports addition and division. The calculator should accept three arguments: two numbers and the operation, either *multiply*, *add* or *divide*, which tells it what to do with the numbers

In JavaScript the code would require additional validation to make sure the last argument is indeed a string. TypeScript offers a way to define specific types for inputs, which describe exactly what type of input is acceptable. On top of that, TypeScript can also show the info of the accepted values already on editor level.

We can create a *type* using the TypeScript native keyword *type*. Let's describe our type *Operation*:

```
type Operation = 'multiply' | 'add' | 'divide';
```

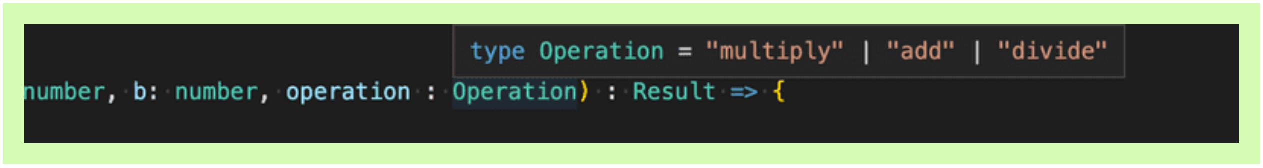
Now the *Operation* type accepts only three kinds of input; exactly the three strings we wanted. Using the OR operator `|` we can define a variable to accept multiple values by creating a union type. In this case we used exact strings (that in technical terms are called string literal types) but with unions you could also make the compiler to accept for example both string and number: `string | number`.

The *type* keyword defines a new name for a type, a type alias. Since the defined type is a union of three possible values, it is handy to give it an alias that has a representative name.

Let's look at our calculator now:

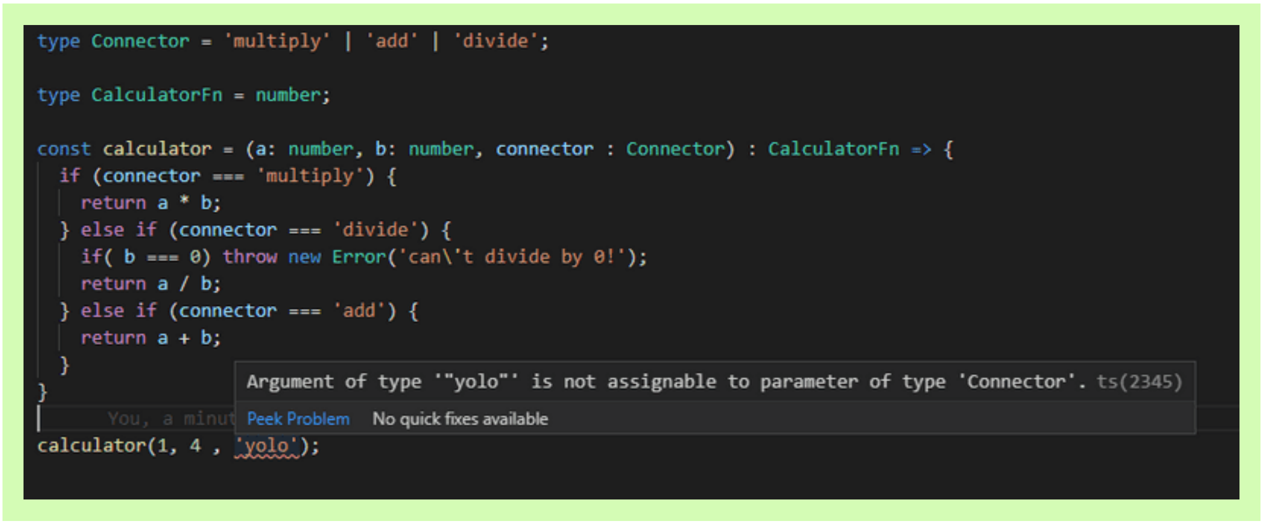
```
type Operation = 'multiply' | 'add' | 'divide';

const calculator = (a: number, b: number, op : Operation) => {
  if (op === 'multiply') {
    return a * b;
  } else if (op === 'add') {
    return a + b;
  } else if (op === 'divide') {
    if (b === 0) return 'can\'t divide by 0!';
    return a / b;
  }
}
```

Now when we hover on top of the *Operation* type in the calculator function, we can immediately see suggestions on what to do with it:



```
                                 type Operation = "multiply" | "add" | "divide"
number, b: number, operation : Operation) : Result => {
```

And if we try to use a value that is not within the *Operation* type, we get the familiar red warning signal and extra info from our editor:



```
type Connector = 'multiply' | 'add' | 'divide';

type CalculatorFn = number;

const calculator = (a: number, b: number, connector : Connector) : CalculatorFn => {
  if (connector === 'multiply') {
    return a * b;
  } else if (connector === 'divide') {
    if( b === 0) throw new Error('can\'t divide by 0!');
    return a / b;
  } else if (connector === 'add') {
    return a + b;
  }
}                 Argument of type '"yolo"' is not assignable to parameter of type 'Connector'. ts(2345)
      You, a minut Peek Problem    No quick fixes available
calculator(1, 4 , 'yolo');
```

This is already pretty nice, but one thing we haven't touched yet is typing the return value of a function. Usually you want to know what a function returns, and it would be nice to have a guarantee that it actually returns what it says it does. Let's add a return value *number* to the calculator function:

```
type Operation = 'multiply' | 'add' | 'divide';

const calculator = (a: number, b: number, op: Operation): number => {

  if (op === 'multiply') {
    return a * b;
  } else if (op === 'add') {
    return a + b;
  } else if (op === 'divide') {
    if (b === 0) return 'this cannot be done';
    return a / b;
  }
}
```

The compiler complains straight away, because in one case the function returns a string. There are couple of ways to fix this: we could extend the return type to allow string values, like so

```
const calculator = (a: number, b: number, op: Operation): number | string =>  {
  // ...
}
```

or we could create a return type which includes both possible types, much like our Operation type

```
type Result = string | number:

const calculator = (a: number, b: number, op: Operation): Result =>  {
  // ...
}
```

But now the question is if it's *really* okay for the function to return a string?

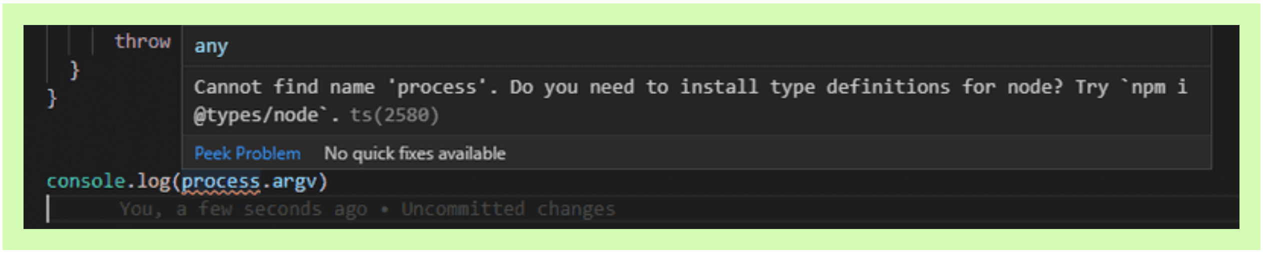
When your code can end up in a situation where something is divided by 0, something has probably gone terribly wrong and an error should be thrown and handled where the function was called. When you are deciding to return values you weren't originally expecting, the warnings you see from TypeScript prevent you from making rushed decisions and help you to keep your code working as expected.

One more thing to consider is, that even though we have defined types for our parameters, the generated JavaScript used at runtime does not contain the type checks. So if for example the *operation*-parameter's value comes from an external interface, there is no definite guarantee that it will be one of the allowed values. Therefore it's still better to include error handling and be prepared for the unexpected to happen. In this case, when there are multiple possible accepted values and all unexpected ones should result in an error, the switch...case statement suits better than if...else in our code.

The code of our calculator should actually look something like this:

```
type Operation = 'multiply' | 'add' | 'divide';
type Result = number;

const calculator = (a: number, b: number, op : Operation) : Result => {
  switch(op) {
    case 'multiply':
      return a * b;
    case 'divide':
      if( b === 0) throw new Error('Can\'t divide by 0!');
      return a / b;
    case 'add':
      return a + b;
    default:
      throw new Error('Operation is not multiply, add or divide!');
  }
}

try {
  console.log(calculator(1, 5 , 'divide'))
} catch (e) {
  console.log('Something went wrong, error message: ', e.message);
}
```

The programs we have written are alright, but it sure would be better if we could use command line arguments instead of always having to change the code to calculate different things.
Let's try it out, as we would in a regular Node application, by accessing *process.argv*. But something is not right:



```
    throw   any
  }         Cannot find name 'process'. Do you need to install type definitions for node? Try `npm i
}           @types/node`. ts(2580)
            Peek Problem   No quick fixes available
console.log(process.argv)
        You, a few seconds ago • Uncommitted changes
```

@types/{npm_package}

Let's return to the basic idea of TypeScript. TypeScript expects all globally used code to be typed, as it does for your own code when your project has a reasonable configuration. The TypeScript library itself contains only typings for the code of the TypeScript package. It is possible to write your own typings for a library, but that is almost never needed - since the TypeScript community has done it for us!

As with npm, the TypeScript world also celebrates open source code. The community is active and continuously reacting to updates and changes in commonly used npm-packages.
You can almost always find the typings for npm-packages, so you don't have to create types for all of your thousands of dependencies alone.

Usually types for existing packages can be found from the *@types*-organization within npm, and you can add the relevant types to your project by installing an npm package with the name of your package with @types/ - prefix. For example: *npm install --save-dev @types/react @types/express @types/lodash @types/jest @types/mongoose* and so on and so on. The *@types/** are maintained by Definitely typed, a community project with the goal to maintaining types of everything in one place.

Sometimes an npm package can also include its types within the code and in that case installing the corresponding *@types/** is not necessary.

> NB: Since the typings are only used before compilation, the typings are not needed in the production build and they should *always* be in the devDependencies of the package.json.

Since the global variable *process* is defined by Node itself, we get its typings by installing the package *@types/node*:

```
npm install --save-dev @types/node
```

After installing the types, our compiler does not complain about the variable *process* anymore. Note that there is no need to require the types to the code, the installation of the package is enough!

Improving the project

Next let's add npm scripts to run our two programs *multiplier* and *calculator*:

```
{
  "name": "part1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "ts-node": "ts-node",
    "multiply": "ts-node multiplier.ts",
    "calculate": "ts-node calculator.ts"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "ts-node": "^8.6.2",
    "typescript": "^3.8.2"
  }
}
```

We can get the multiplier to work with command line parameters with the following changes

```
const multiplicator = (a: number, b: number, printText: string) => {
  console.log(printText,  a * b);
}

const a: number = Number(process.argv[2])
const b: number = Number(process.argv[3])
multiplicator(a, b, `Multiplied ${a} and ${b}, the result is:`);
```

and we can run it with

```
npm run multiply 5 2
```

if the program is run with parameters that are not of the right type, e.g.

```
npm run multiply 5 lol
```

it "works" but gives us the answer

```
Multiplied 5 and NaN, the result is: NaN
```

The reason for this is, that *Number('lol')* returns *NaN*, which is actually type *number*, so TypeScript has no power to rescue us from this kind of situation.

In order to prevent this kind of behaviour, we have to validate the data given to us from the command line.

Improved version of the multiplicator looks like this:

```
interface MultiplyValues {
  value1: number;
  value2: number;
}

const parseArguments = (args: Array<string>): MultiplyValues => {
  if (args.length < 4) throw new Error('Not enough arguments');
  if (args.length > 4) throw new Error('Too many arguments');

  if (!isNaN(Number(args[2])) && !isNaN(Number(args[3]))) {
    return {
      value1: Number(args[2]),
      value2: Number(args[3])
    }
  } else {
    throw new Error('Provided values were not numbers!');
  }
}

const multiplicator = (a: number, b: number, printText: string) => {
  console.log(printText,  a * b);
}

try {
  const { value1, value2 } = parseArguments(process.argv);
  multiplicator(value1, value2, `Multiplied ${value1} and ${value2}, the result is:`);
} catch (e) {
  console.log('Error, something bad happened, message: ', e.message);
}
```

When we now run the program

```
npm run multiply 1 lol
```

we get a proper error message:

```
Error, something bad happened, message:  Provided values were not numbers!
```

Definition of the function *parseArguments* has a couple of interesting things:

```
const parseArguments = (args: Array<string>): MultiplyValues => {
  // ...
}
```

Firstly, the parameter *args* is an array of strings. The return value has the type *MultiplyValues*, which is defined as follows:

```
interface MultiplyValues {
  value1: number;
  value2: number;
}
```

The definition utilizes TypeScript's Interface keyword, which is one way to define the "shape" an object should have. In our case it is quite obvious that the return value should be an object with two properties *value1* and *value2*, which should both be of type number.

Exercises 9.1.-9.3.

setup

Exercises 9.1.-9.7. will be all made to the same node project. Create the project in an empty directory with *npm init* and install the ts-node and typescript packages. Create also the file *tsconfig.json* to the directory with the following content:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
  }
}
```

The *tsconfig.json* file is used to define how the TypeScript compiler should interpret the code, how strictly the compiler should work, which files to watch or ignore, and much much more. For now we will only use the compiler option noImplicitAny, that makes it mandatory to have types for all variables used.

9.1 Body mass index

Create the code of this exercise to file *bmiCalculator.ts*

Write a function *calculateBmi* that counts BMI based on given height (in centimeters) and weight (in kilograms) and then returns a message that suits the results.

Call the function in the same file with hard-coded parameters and print out the result. The code

```
console.log(calculateBmi(180, 74))
```

should print the following message

```
Normal (healthy weight)
```

Create a npm script for running the program with command *npm run calculateBmi*

9.2 Exercise calculator

Create the code of this exercise to file *exerciseCalculator.ts*

Write a function *calculateExercises* that calculates the average time of *daily exercise hours* and compares it to the *target amount* of daily hours and returns an object that includes the following values:

- the number of days
- the number of training days
- the original target value
- the calculated average time
- boolean value describing if the target was reached
- a rating between the numbers 1-3 that tells how well the hours are met. You can decide on the metric on your own.
- a text value explaining the rating

The daily exercise hours are given to the function as an array that contains the number of exercise hours for each day in the training period. Eg. a week with 3 hours of training on Monday, none on Tuesday, 2 hours on Wednesday, 4.5 hours on Thursday and so on would be represented by the following array:

```
[3, 0, 2, 4.5, 0, 3, 1]
```

For the Result object you should create an interface.

If you would call the function with parameters *[3, 0, 2, 4.5, 0, 3, 1]* and *2* it could return

```
{ periodLength: 7,
  trainingDays: 5,
  success: false,
  rating: 2,
  ratingDescription: 'not too bad but could be better',
  target: 2,
  average: 1.9285714285714286 }
```

Create a npm script *npm run calculateExercises* for calling the function with hard coded values.

9.3 Command line

Change the previous exercises so that you can give the parameters of *bmiCalculator* and *exerciseCalculator* as command line arguments.

Your program could work eg. as follows:

```
$ npm run calculateBmi 180 91

Overweight
```

and

```
$ npm run calculateExercises 2 1 0 2 4.5 0 3 1 0 4

{ periodLength: 9,
  trainingDays: 6,
  success: false,
  rating: 2,
  ratingDescription: 'not too bad but could be better',
  target: 2,
  average: 1.7222222222222223 }
```

In the example the *first argument* is the target value.

Handle exceptions and errors appropriately. The exerciseCalculator should accept inputs of varied lengths. Determine by yourself how you manage to collect all needed input.

More about tsconfig

In the exercises we used only one tsconfig rule noImplicitAny. It's a good place to start, but now it's time to look into the config file a little deeper.

The tsconfig.json file contains all your core configurations on how you want TypeScript to work in your project. You can define how strictly you want the code to be inspected, what files to include and exclude (*node_modules* is excluded by default), and where compiled files should be placed (more on this later).

Let's specify the following configurations in our *tsconfig.json* file:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true,
    "moduleResolution": "node"
  }
}
```

Do not worry too much about the *compilerOptions*, they will be under closer inspection later on.

You can find explanations for each of the configurations from the TypeScript documentation, or the really handy tsconfig page, or from the tsconfig schema definition, which unfortunately is formatted a little worse than the first two options.

Adding express to the mix

Right now we are at a pretty good place. Our project is set up and we have two executable calculators in it. However, since our aim is to learn FullStack development, it is time to start working with some HTTP-requests.

Let us start by installing express:

```
npm install express
```

add then add the *start* script to package.json:

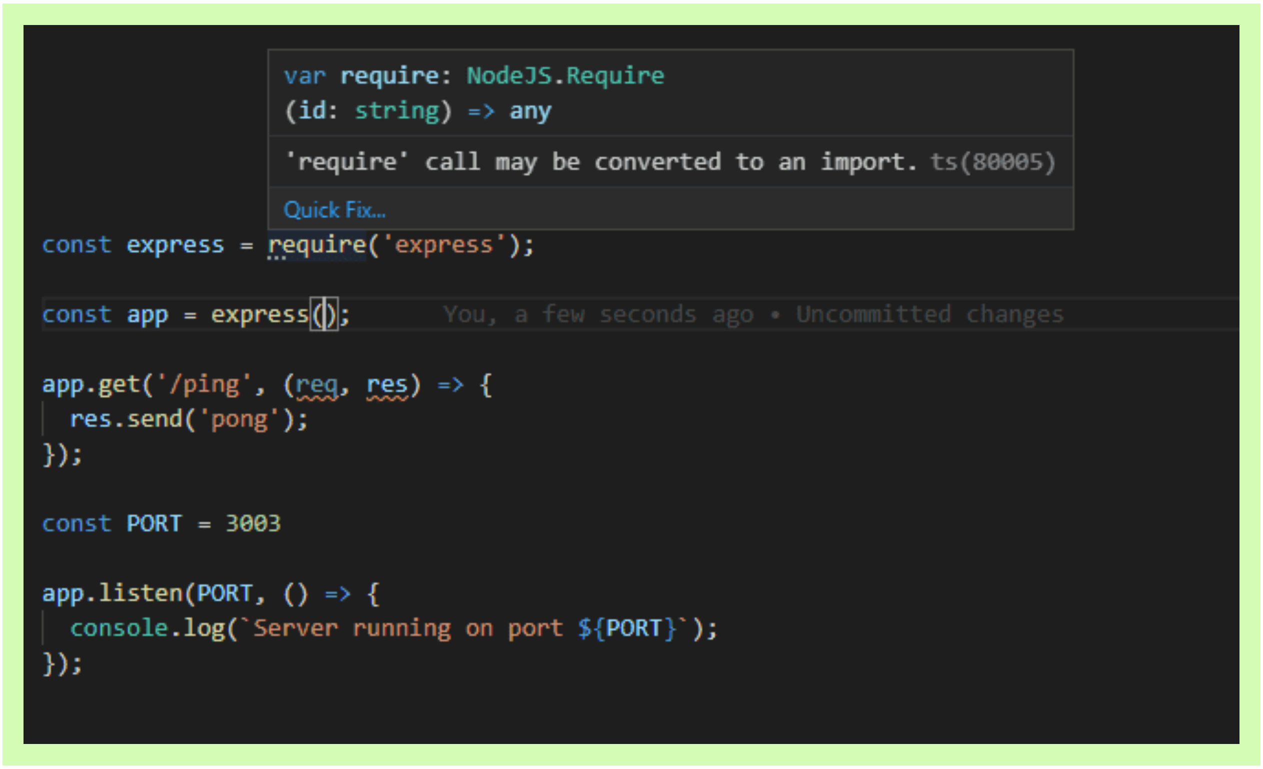
```
{
  // ..
  "scripts": {
    "ts-node": "ts-node",
    "multiply": "ts-node multiplier.ts",
    "calculate": "ts-node calculator.ts",
    "start": "ts-node index.ts"
  },
  // ..
}
```

Now we can create the file *index.ts*, and write the HTTP GET *ping* endpoint to it:

```
const express = require('express');
const app = express();

app.get('/ping', (req, res) => {
  res.send('pong');
});
```

```
const PORT = 3003;

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Everything else seems to be working just fine, but as you'd expect the *req* and *res* parameters of *app.get* need typing. If you look carefully, VSCode is also complaining something about importing express. You can see a short yellow line of dots under the *require*. Let's hover over the problem:



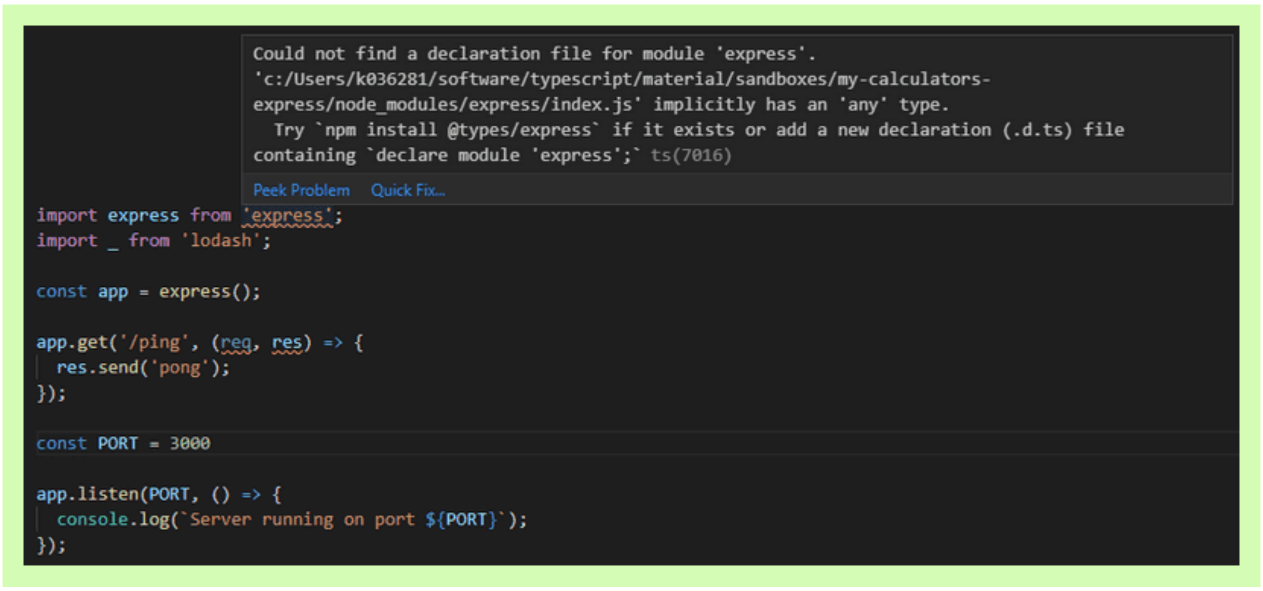
```
var require: NodeJS.Require
(id: string) => any
'require' call may be converted to an import. ts(80005)
Quick Fix...
const express = require('express');

const app = express();        You, a few seconds ago • Uncommitted changes

app.get('/ping', (req, res) => {
  res.send('pong');
});

const PORT = 3003

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

The complaint is that the *'require' call may be converted to an import*. Let us follow the advice and write the import as follows

```
import express from 'express';
```

NB: VSCode offers you a possibility to fix the issues automatically by clicking the *Quick fix...* button. Keep your eyes open for these helpers/quick fixes; listening to your editor usually makes your code better and easier to read. The automatic fixes for issues can be a major time saver as well.

Now we run into another problem - the compiler complains about the import statement. Once again the editor is our best friend when trying to find out what the issue is:



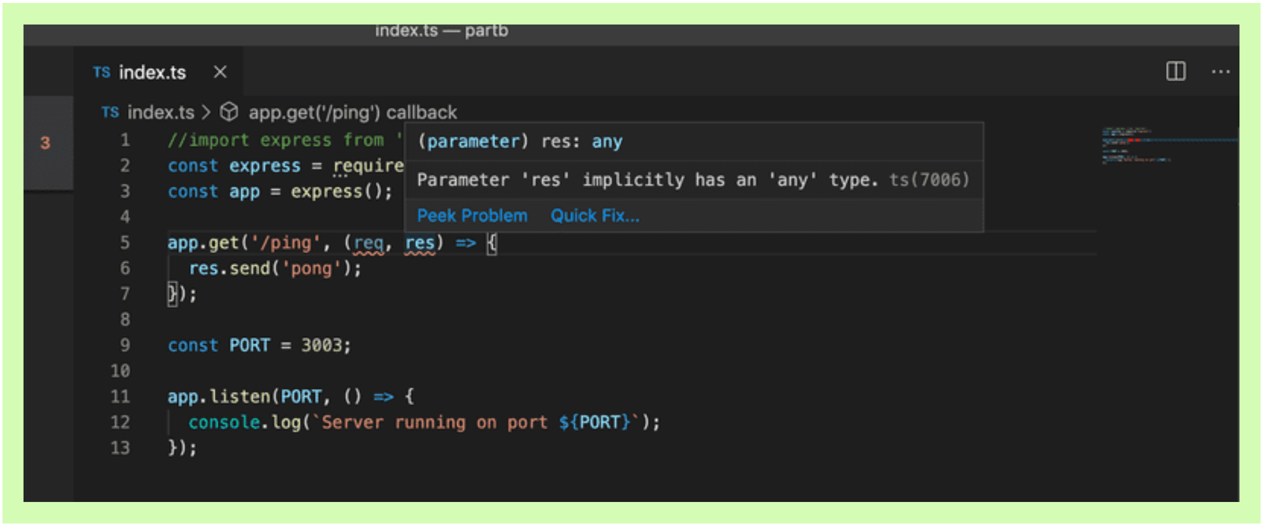
```
Could not find a declaration file for module 'express'.
'c:/Users/k036281/software/typescript/material/sandboxes/my-calculators-
express/node_modules/express/index.js' implicitly has an 'any' type.
  Try `npm install @types/express` if it exists or add a new declaration (.d.ts) file
containing `declare module 'express';` ts(7016)
Peek Problem    Quick Fix...
import express from 'express';
import _ from 'lodash';

const app = express();

app.get('/ping', (req, res) => {
  res.send('pong');
});

const PORT = 3000

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

We haven't installed types for *express*. Let's do what the suggestion says and run:

```
npm install --save-dev @types/express
```

And no more errors! Let's take a look at what changed.

When we hover over the *require* statement, we can see the compiler interprets everything express related to be of type *any*.



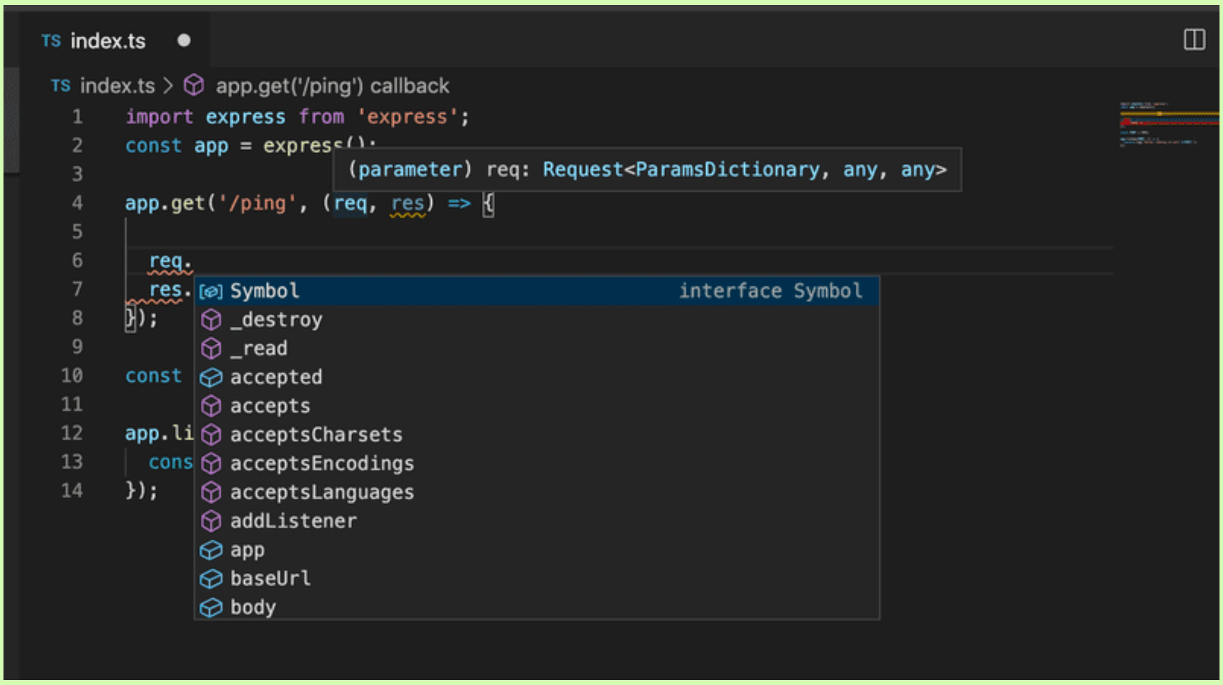
index.ts — partb

index.ts > app.get('/ping') callback

```
//import express from '
const express = require
const app = express();

app.get('/ping', (req, res) => {
  res.send('pong');
});

const PORT = 3003;

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

```
(parameter) res: any
Parameter 'res' implicitly has an 'any' type. ts(7006)
Peek Problem    Quick Fix...
```

Whereas when we use *import*, the editor knows the actual types



```
import express from 'express';
const app = express();

app.get('/ping', (req, res) => {

  req.
  res.
});

const

app.li
  cons
});
```

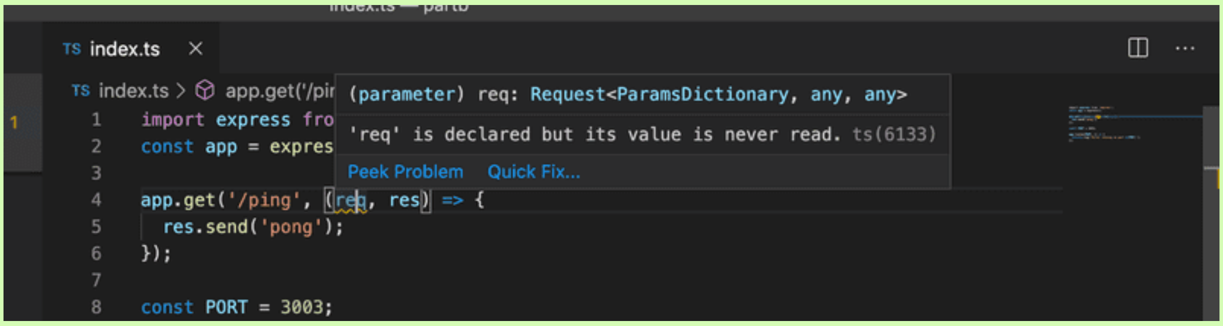
```
(parameter) req: Request<ParamsDictionary, any, any>
```

Which import statement to use depends on the export method used in the imported package.

A good rule of thumb is to try importing a module using the *import* statement first. We will always use this method in the *frontend*. If *import* does not work, try a combined method: *import ... = require('...')*.

We strongly suggest you read more about TypeScript modules here.

There is one more problem with the code



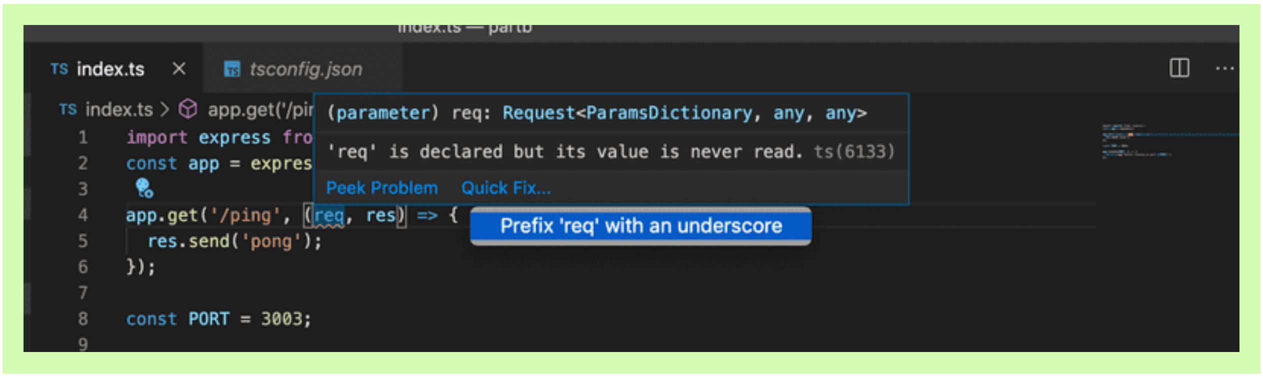
```
import express from 'express';
const app = express();

app.get('/ping', (req, res) => {
  res.send('pong');
});

const PORT = 3003;
```

```
(parameter) req: Request<ParamsDictionary, any, any>
'req' is declared but its value is never read. ts(6133)
```

This is because we banned unused parameters in our *tsconfig.json*

```
{
  "compilerOptions": {
    "target": "ES2020",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true
    }
  }
```

This configuration might create problems if you have library-wide predefined functions which require declaring a variable even if it's not used at all, as is the case here. Fortunately this issue has already been solved on configuration level. Once again hovering over the issue gives us a solution. This time we can just click the quick fix button:



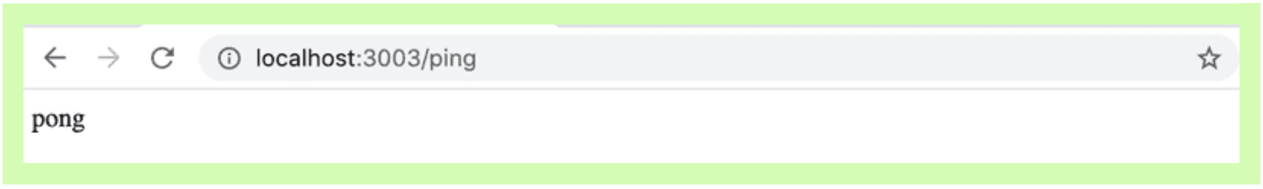
```
import express fro
const app = expres

app.get('/ping', (req, res) => {
  res.send('pong');
});

const PORT = 3003;
```

```
(parameter) req: Request<ParamsDictionary, any, any>
'req' is declared but its value is never read. ts(6133)
```

If it absolutely impossible to get rid of an unused variable, you can prefix it with an underscore to inform the compiler you have thought about it and there is nothing you can do.

Let's rename the *req* variable to *_req*.

Finally we are ready to start the application. It seems to work fine:



pong

To simplify the development we should enable *auto reloading* to improve our workflow. In this course you have already used *nodemon*, but ts-node has an alternative called *ts-node-dev*. It is meant to be used only with a development environment which takes care of recompilation on every change, so restarting the application won't be necessary.

Let's install *ts-node-dev* to our development dependencies

```
npm install --save-dev ts-node-dev
```

add a script to *package.json*

```
{
  // ...
  "scripts": {
      // ...
```

```
    "dev": "ts-node-dev index.ts",
  },
  // ...
}
```

And now by running *npm run dev* we have a working, auto-reloading development environment for our project!

Exercises 9.4.-9.5.

9.4 Express

Add express to your dependencies and create a HTTP GET endpoint *hello* that answers 'Hello Full Stack!'

The web app should be started with command *npm start* in production mode and *npm run dev* in development mode that should use *ts-node-dev* to run the app.

Replace also your existing *tsconfig.json* file with the following content:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "noImplicitReturns": true,
    "strictNullChecks": true,
    "strictPropertyInitialization": true,
    "strictBindCallApply": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitThis": true,
    "alwaysStrict": true,
    "esModuleInterop": true,
    "declaration": true,
  }
}
```

make sure there are not any errors!

9.5 WebBMI

Add an endpoint for BMI-calculator that can be used by doing a HTTP GET request to endpoint *bmi* and specifying the input with query string parameters. For example to get bmi for a person having height 180 and weight 72, the url is http://localhost:3002/bmi?height=180&weight=72

The response is a json of the form

```
{
  weight: 72,
  height: 180,
  bmi: "Normal (healthy weight)"
}
```

See the express documentation for info how to access the query parameters.

If the query parameters of the request are of the wrong type or missing, response with proper status code and error message are given

```
{
  error: "malformatted parameters"
}
```

Do not copy the calculator code to file *index.ts*, make it a typescript module that can be imported in *index.ts*.

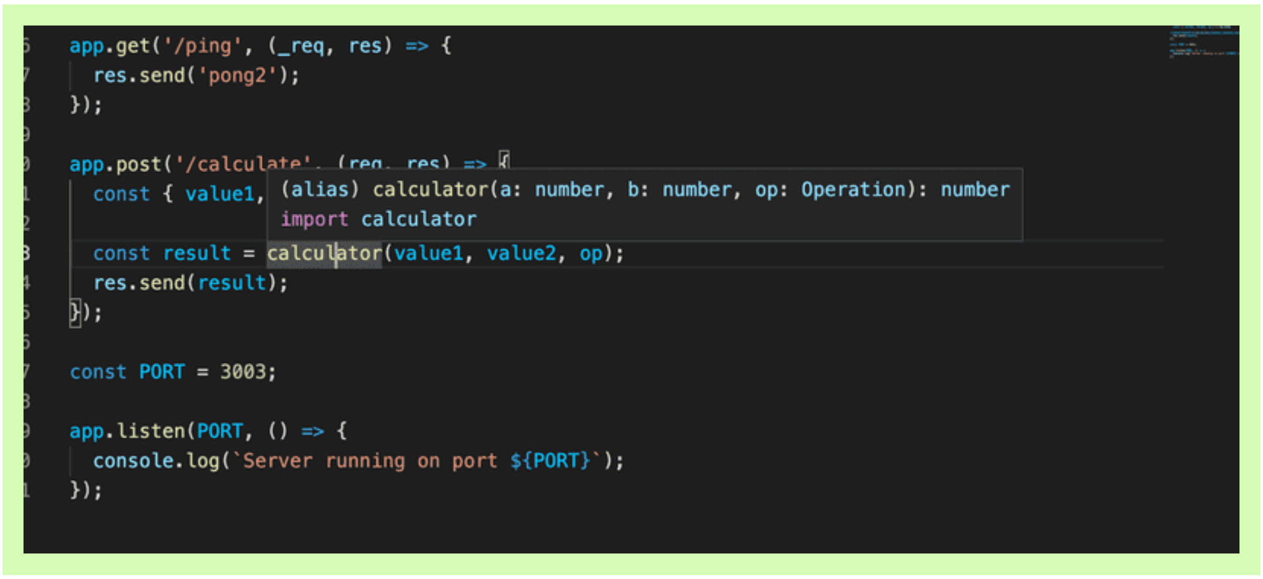
The horrors of *any*

Now that we have our first endpoints completed, you might notice we have used barely any TypeScript in these small examples. When examining the code a bit closer, we can see a few dangers lurking there.

Let's add an HTTP POST endpoint *calculate* to our app:

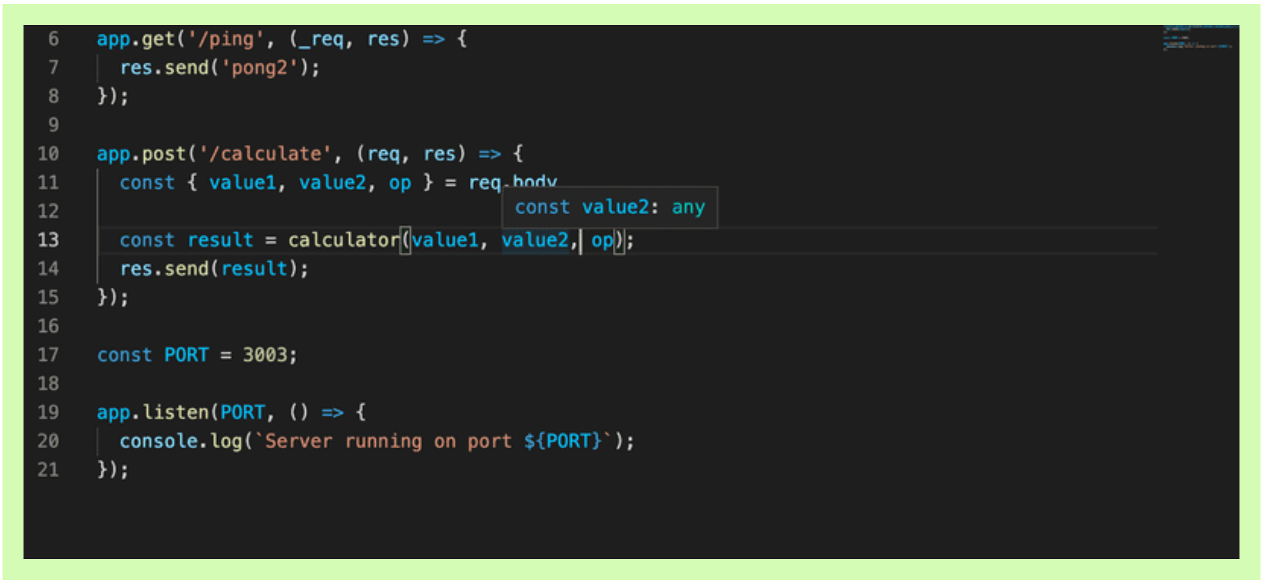
```
import { calculator } from './calculator'

// ...

app.post('/calculate', (req, res) => {
  const { value1, value2, op } = req.body

  const result = calculator(value1, value2, op);
  res.send(result);
});
```

When you hover over the *calculate* function, you can see the typing of the *calculator* even though the code itself does not contain any typings:



```
app.get('/ping', (_req, res) => {
  res.send('pong2');
});

app.post('/calculate', (req, res) => {
  const { value1, (alias) calculator(a: number, b: number, op: Operation): number
                  import calculator
  const result = calculator(value1, value2, op);
  res.send(result);
});

const PORT = 3003;

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

But if you hover over the values parsed from the request, an issue arises:



```
app.get('/ping', (_req, res) => {
  res.send('pong2');
});

app.post('/calculate', (req, res) => {
  const { value1, value2, op } = req.body
                                const value2: any
  const result = calculator(value1, value2, op);
  res.send(result);
});

const PORT = 3003;

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

All of the variables have type *any*. It is not all that surprising, as no one has given them a type yet. There are a couple of ways to fix this, but the first we have to consider why this is accepted and where did the type *any* come from?

In TypeScript every untyped variable whose type cannot be inferred, becomes implicitly `any` type. Any is a kind of a "wild card" type which literally stands for *whatever type*. Things become implicitly any type quite often when one forgets to type functions.

We can also explicitly type things *any*. The only difference between implicit and explicit any type is how the code looks, the compiler does not care about the difference.

Programmers however see the code differently when *any* is explicitly enforced than when it implicitly inferred. Implicit *any* typings are usually considered problematic, since it is quite often due to the coder forgetting to assign types (or being too lazy to do it), and it also means that the full power of TypeScript is not properly exploited.

This is why the configuration rule `noImplicitAny` exists on compiler level, and it is highly recommended to keep it on at all times. In the rare occasions you seriously cannot know what the type of a variable is, you should explicitly state that in the code

```
const a : any = /* no clue what the type will be! */.
```

We already have *noImplicitAny* configured in our example, so why does the compiler not complain about the implicit *any* types? The reason is, that the *query* field of an express Request object is explicitly typed *any*. Same is true for the *request.body* field we use to post data to an app.

What if we would like to prevent developers from using *any* type at all? Fortunately we have other methods than *tsconfig.json* to enforce coding style. What we can do is use *eslint* to manage our code. Let's install eslint and its TypeScript extensions:

```
npm install --save-dev eslint @typescript-eslint/eslint-plugin @typescript-eslint/parser
```

We will configure eslint to disallow explicit any. Write the following rules to *.eslintrc*:

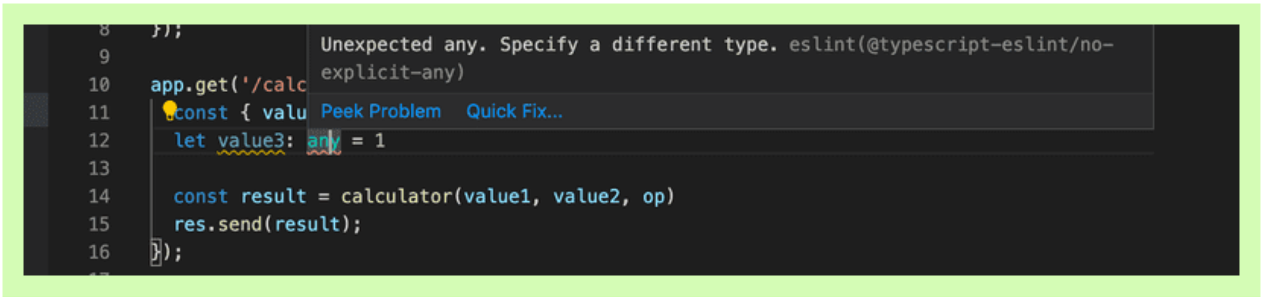
```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 11,
    "sourceType": "module"
  },
  "plugins": ["@typescript-eslint"],
  "rules": {
    "@typescript-eslint/no-explicit-any": 2
  }
}
```

(Newer versions of eslint has this rule on by default, so you don't necessarily need to add it separately.)

Let us also set up a *lint* npm script to inspect the files with *.ts* extension by modifying the *package.json* file:

```
{
  // ...
  "scripts": {
      "start": "ts-node index.ts",
      "dev": "ts-node-dev index.ts",
      "lint": "eslint --ext .ts ."
      //  ...
  },
  // ...
}
```

Now lint will complain if we try to define a variable of type *any*:



```
});

app.get('/calc
  const { valu
  let value3: any = 1

  const result = calculator(value1, value2, op)
  res.send(result);
});
```

```
Unexpected any. Specify a different type. eslint(@typescript-eslint/no-explicit-any)
```

The @typescript-eslint has a lot of TypeScript specific eslint rules, but you can also use all basic eslint rules in TypeScript projects. For now we should probably go with the recommended settings and modify the rules as we go along whenever we find something we want to behave differently.

On top of the recommended settings, we should try to get familiar with the coding style required in this part and *set the semicolon at the end of each line of code to required*.

So we will use the following *.eslintrc*

```
{
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking"
  ],
  "plugins": ["@typescript-eslint"],
  "env": {
    "node": true,
    "es6": true
  },
  "rules": {
    "@typescript-eslint/semi": ["error"],
    "@typescript-eslint/explicit-function-return-type": "off",
    "@typescript-eslint/explicit-module-boundary-types": "off",
    "@typescript-eslint/restrict-template-expressions": "off",
    "@typescript-eslint/restrict-plus-operands": "off",
    "@typescript-eslint/no-unused-vars": [
      "error",
      { "argsIgnorePattern": "^_" }
    ],
    "no-case-declarations": "off"
  },
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json"
  }
}
```

There are quite a few semicolons missing, but those are easy to add.

And now let's fix everything that needs to be fixed!

Exercises 9.6.-9.7.

9.6 Eslint

Configure your project to use the above eslint settings and fix all the warnings.

9.7 WebExercises

Add an endpoint to your app for the exercise calculator. It should be used by doing a HTTP POST request to endpoint *exercises* with the input in the request body

```
{
  "daily_exercises": [1, 0, 2, 0, 3, 0, 2.5],
  "target": 2.5
}
```

Response is a json of the following form

```
{
    "periodLength": 7,
    "trainingDays": 4,
    "success": false,
    "rating": 1,
    "ratingDescription": "bad",
    "target": 2.5,
    "average": 1.2142857142857142
}
```

About course

Course contents

FAQ

Partners

Challenge

If the body of the request is not of the right form, response with proper status code and error message is given. The error message is either

```
{
  error: "parameters missing"
}
```

or

```
{
  error: "malformatted parameters"
}
```

depending on the error. The latter happens if the input values do not have the right type, i.e. they are not numbers or convertable to numbers.

In this exercise you might find it beneficial to use the *explicit any* type when handling the data in the request body. Our eslint configuration is preventing this but you may unset this rule for a particular line by inserting the following comment as the previous line:

```
// eslint-disable-next-line @typescript-eslint/no-explicit-any
```

You might also get in trouble with rules *no-unsafe-member-access* an *no-unsafe-assignment* .

These rules may be ignored in this exercise.

Note that you need to have a correct setup in order to get hold to the request body, see part 3.

Propose changes to material

Part 9a
Previous part

Part 9c
Next part

HOUSTON