Node with TypeScript

https://github.com/peerberger/TypeScriptNode

Demo project: https://github.com/peerberger/TypeScriptNode-menu-api

- This project was built according to this guide, which you can read through to really understand the code.

Creating a Node project with TypeScript

- Open the terminal and run the following commands.
  Ex:

  npm init -y
  npm install --save-dev typescript
  npx tsc --init

Using TypeScript in Node

- Let's run a ts program with node!
  Create a new folder named "**src**", and in it create a file named "**index.ts**".
  In it, write the following.
  Ex:

```
console.log("hello from ts");
```

- Now as you might remember, to run a ts program, we first need to convert it to js.
  To do that, go to package.json, and under scripts, delete the following line.
  Ex:

```
"test": "echo \"Error: no test specified\" && exit 1"
```

  And write the following line in its place.
  Ex:

```
"build": "tsc"
```

- Open the terminal and run the following command.
  Ex:

  npm run-script build

- It will generate a file named "**index.js**" based on the ts code in **index.ts**, and you should see it in the same folder.

- Now you can run the js code with the following line.
  Ex:

node src/index

- But doing all of this is annoying, we can make it much simpler.
  Run the following command to install the **ts-node** package.
  Ex:

  npm install --save-dev ts-node

- With it, we can run a single command on a ts file, which will compile it into js, and launch it on node.
  So let's configure this command in package.json - add the following line under the "scripts" section so it looks like this.
  Ex:

```
"scripts": {
    "start": "ts-node src/index.ts",
    "build": "tsc"
},
```

- To run this script, run the following command.
  The terminal should print the message we wrote in the **index.ts**.
  Ex:

  npm start

- Now, if we want it to execute every time we change the code automatically, we can install the **ts-node-dev** package, which will execute on save.
  It also significantly decreases the time it takes to restart your application when you make a change.
  To install it, run the following command.
  Ex:

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

  And edit the script we added to look like this.
  Ex:

```
"scripts": {
    "start": "ts-node-dev --respawn src/index.ts",
    "build": "tsc"
},
```

  This package doesn't always work on Windows computers, but you can also use the **nodemon** package, it's also more popular.
  Ex:

npm install --save-dev nodemon

```
"scripts": {
    "start": "nodemon --exec ts-node src/index.ts",
    "build": "tsc"
},
```

- Now, we're gonna want to interact with other libraries. We can easily do that with ts libraries, but it can be a little tricky with js libraries.
  For example, we want to use the js library **express** (to install, run the following command).
  Ex:

  npm install express

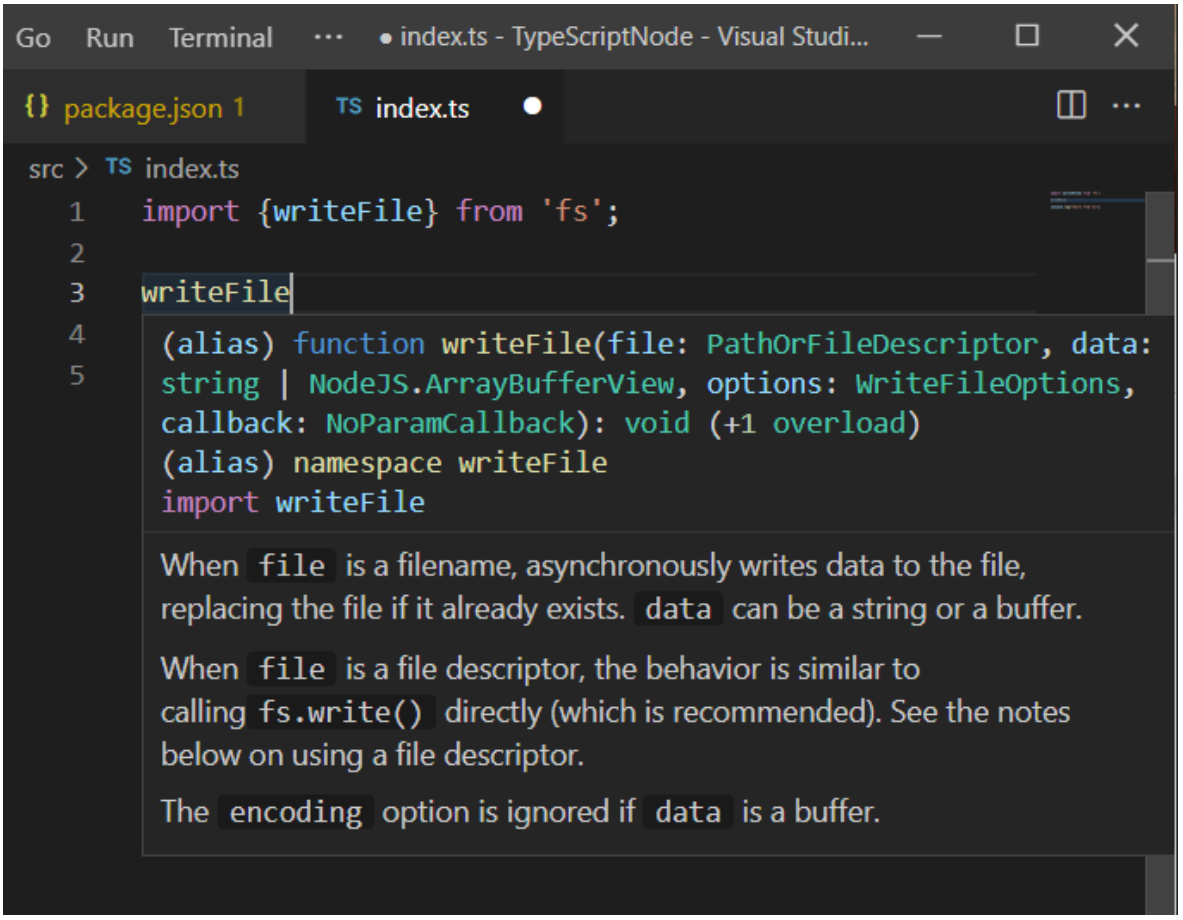
  We can just use it in our ts code, but the whole point of ts is knowing the types and function signatures - but they don't exist in the js library.
  So we need to install another package that contains them.
  Ex:

  npm install --save-dev @types/express

- There's a package like this for node too.
  Ex:

  npm install --save-dev @types/node

  For example, now when you use the **writeFile** function from the **fs** module, you'll see the signature and all this additional info.

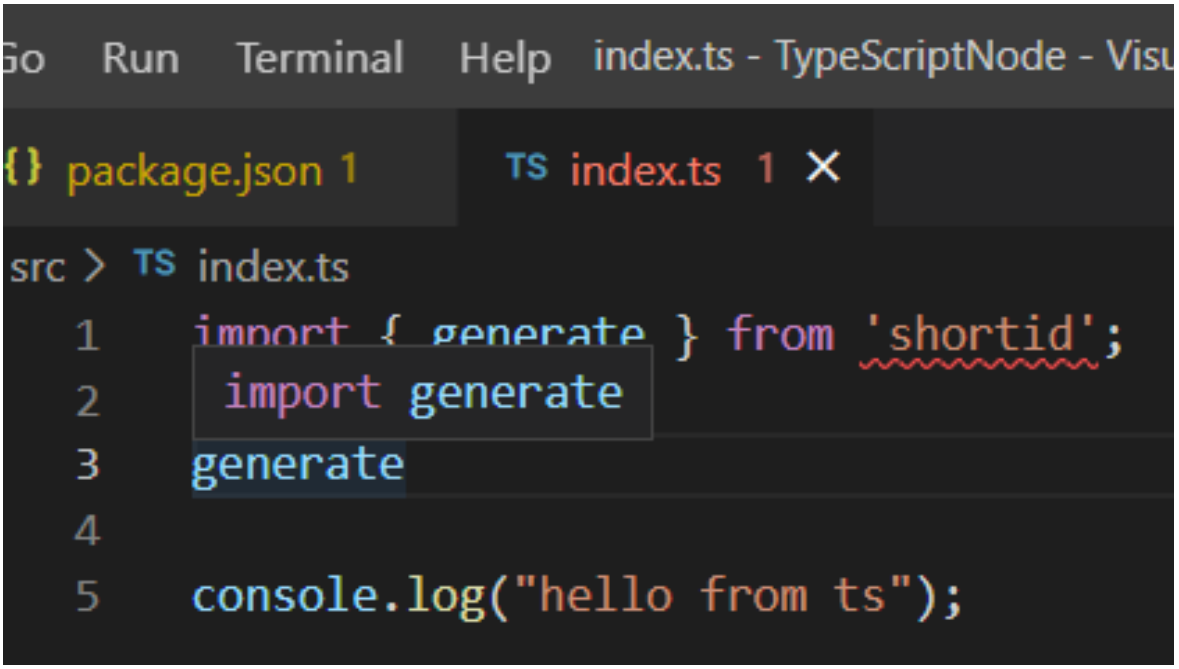


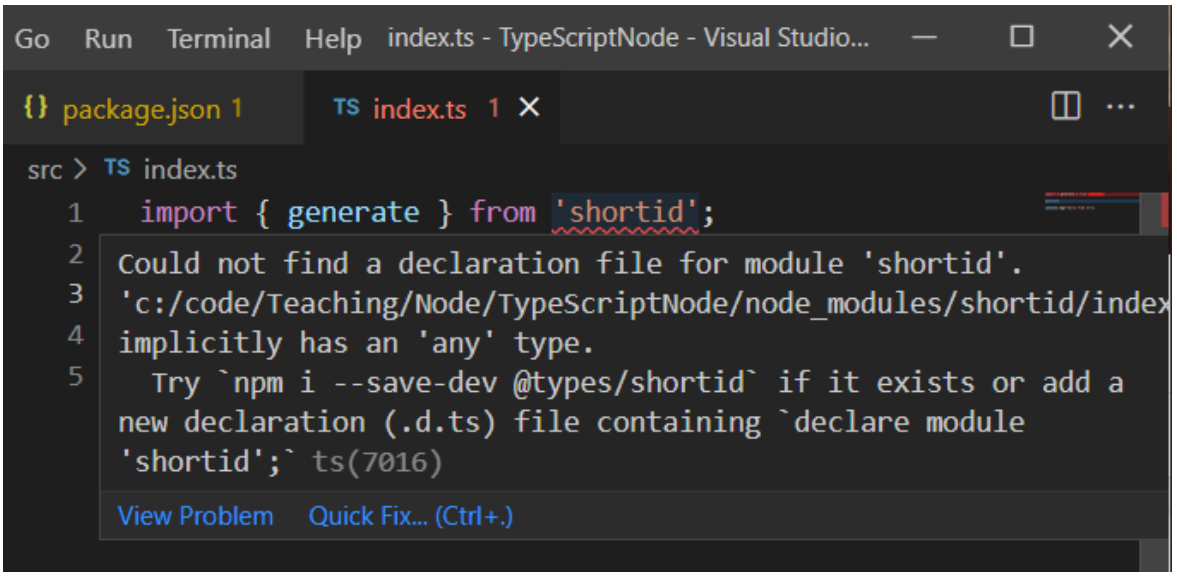
- So from now on, when you install a new js package, you're gonna want to install a **@types** package for it too (their name is usually formatted like this: "**@types/[name of the js package]**").

- But some packages don't have a **@types** package, what do we do then?
  Let's take for example the **shortid** package (it actually does have a @types package, but it's just an example). To install, run the following command.
  Ex:

  npm install shortid

- This is how it looks when you import it in ts.
  You can see it's yelling at you because there's no declaration file for **shortid**.
  Ex:





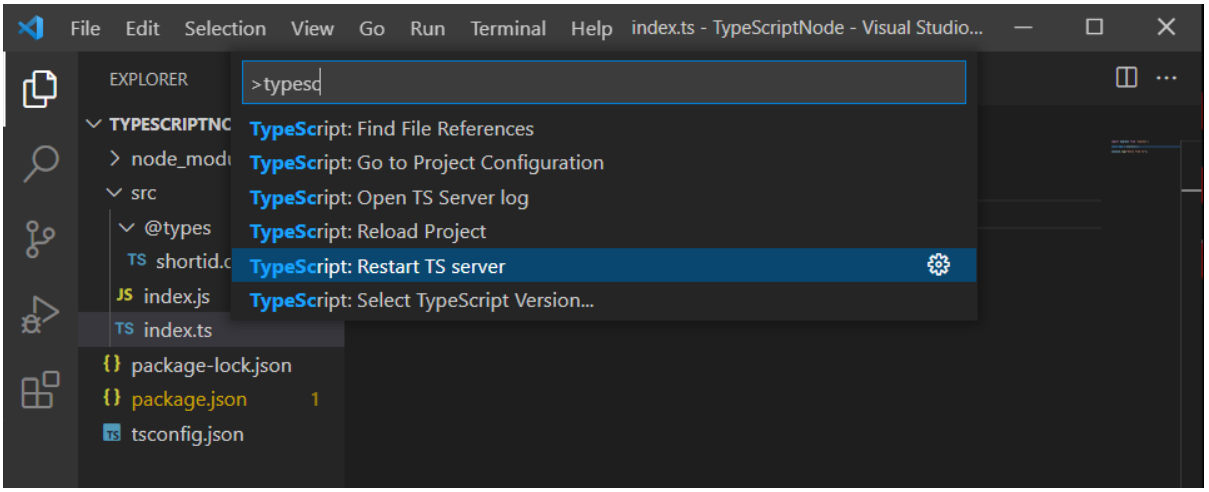
- To deal with that, create a folder named “@types”, and in it create a file named “shortid.d.ts”.
  Then write the following.
  Ex:

```
declare module 'shortid';
```

  Now you can see in index.ts that the error is gone.

- You can theoretically declare the full types definitions of the package, but most packages already have a **@types** package.
  This solution is generally for the rare occasion you come across a package that doesn’t have one, and you need to eliminate the error.

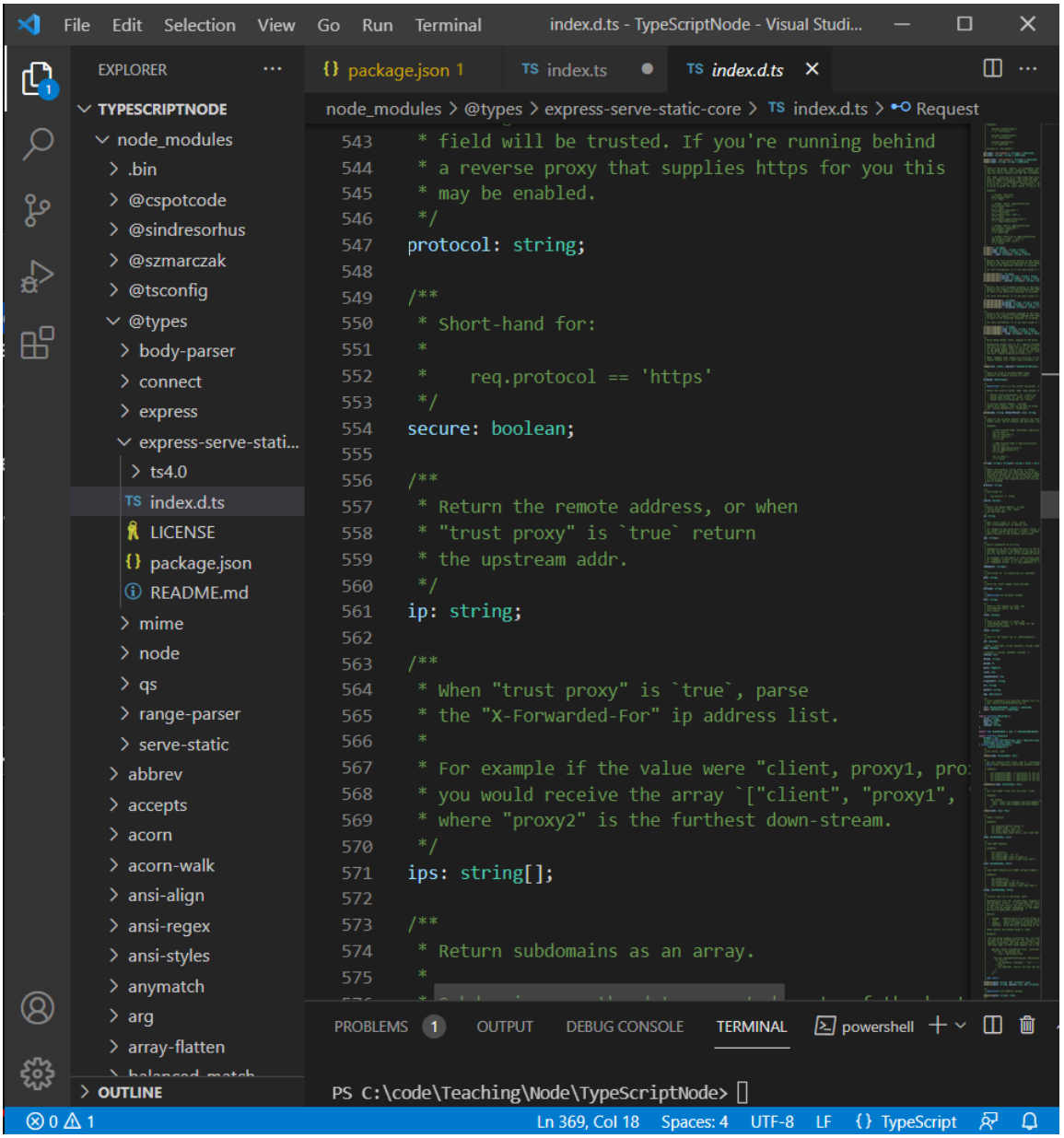
- Another troubleshooting tip is that sometimes the ts compiler is being laggy or slow, and might show you errors like “Duplicate identifier ‘[package name]’.”.
  In these cases you can hit **ctrl+shift+p**, and run **TypeScript: Restart TS Server**.
  This will restart the ts compiler, and hopefully eliminate the errors.



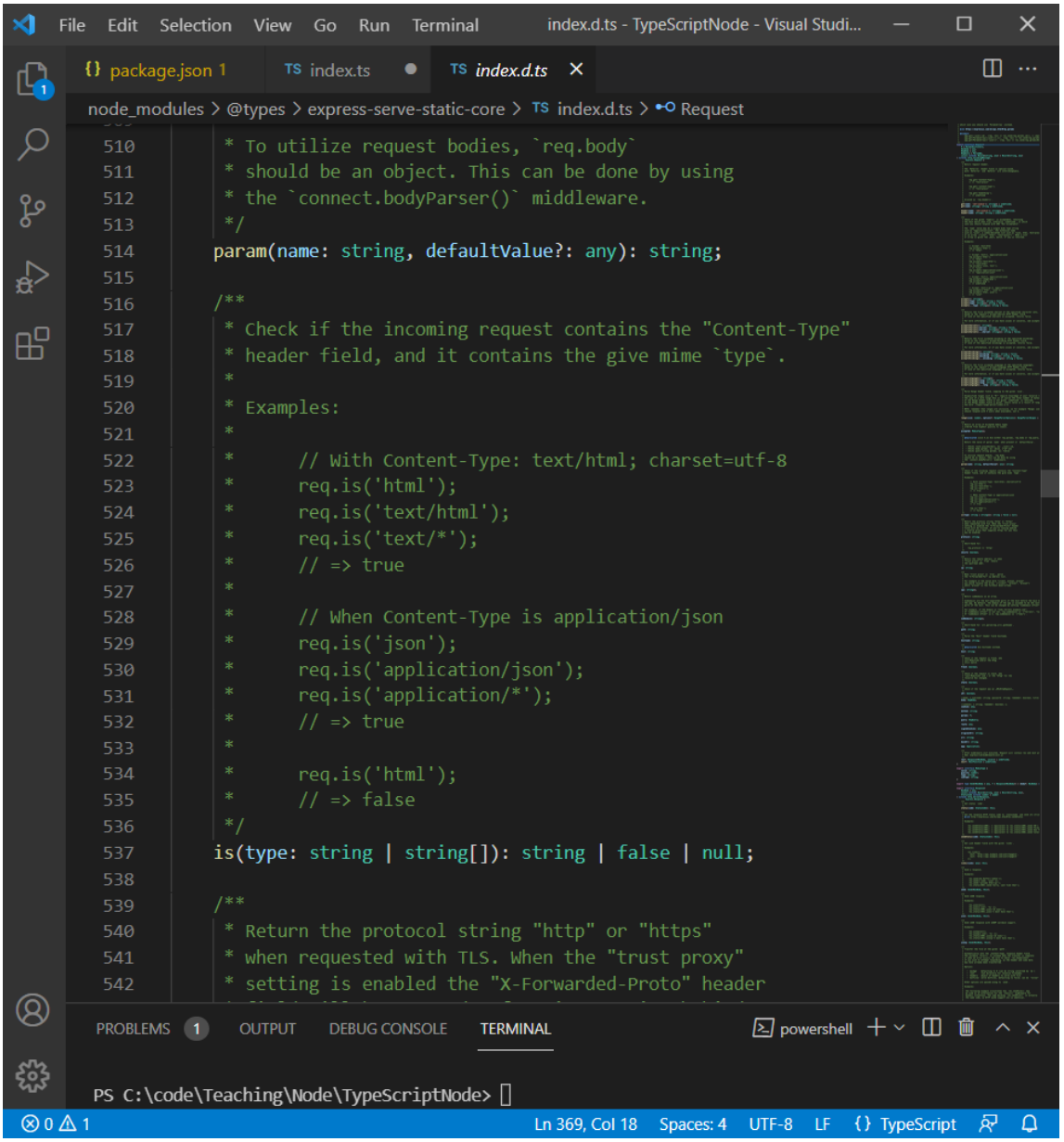
- Now let’s see some practical ts magic.
  Let’s write some simple **express** code.
  Ex:

```
import express from 'express';

const app = express();

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

});

app.listen(3000, () => {
    console.log("started");
});
```

- If you right click on the **req** variable, and click on **Go to Type Definition**, it’ll take you to the definition of the **Request** interface, which has all the type definitions, function signatures, and documentation of **Request**.
  You can see that the **secure** field is of type **boolean**, the **ip** field is of type **string**, etc.

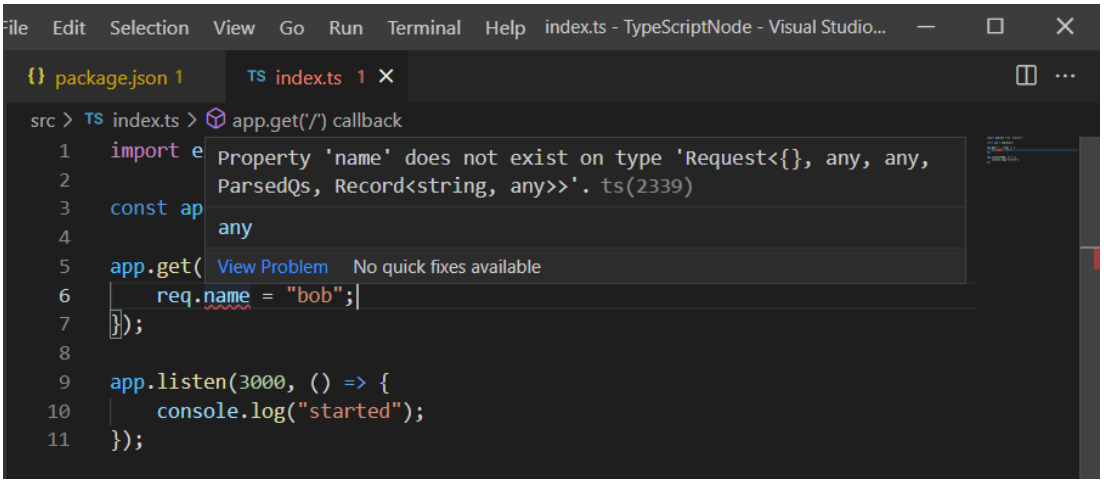


And that the **is()** function receives a **parameter** of either type **string** or **string[]**, and **returns** a value of either type **string**, **false**, or **null**.
You even get examples of implementation in the function's doc comment.



Whereas when you use plain js, you'd have to go to the docs to see all that.

- Next thing in working with ts, is that sometimes you're gonna want to escape the restrictions of ts.
  For example, say you want to assign a new field named "**name**" on **req**.
  You'll get the following error, because ts is all about strong types, and **Request** doesn't have a field named "**name**".
  Ex:



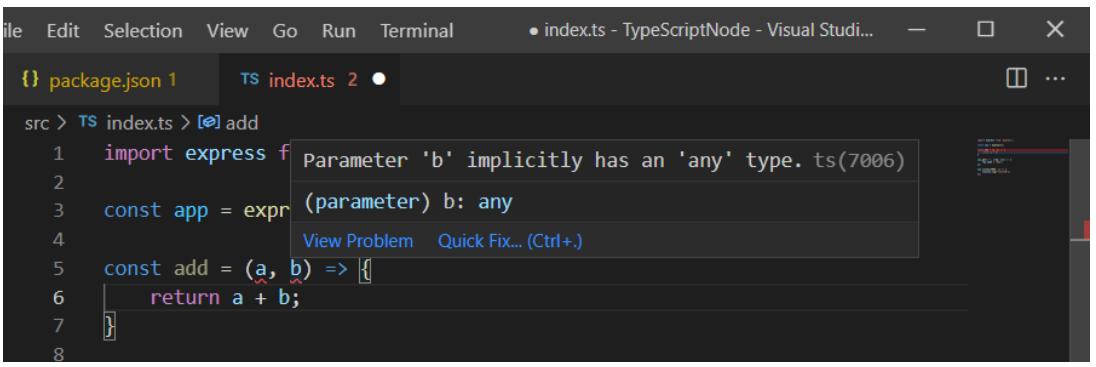
- You can work around that by casting **req** to type **any** (or any type you want) just for that one line.
  Ex:

```
app.get('/', (req) => {
    (req as any).name = "bob";
});
```

  Or you can also cast **req** for the entire scope, by declaring its type in the parameter declaration.
  Ex:

```
app.get('/', (req: any) => {
    req.name = "bob";
});
```

- You should keep an eye on **any** and **undefined**, because ts always wants to know what type a variable is.
  For example, you can't define a function like this, because ts doesn't know the types of the params.
  Ex:



- To fix this, you need to declare the types of the params, like we saw in the previous section.
  Ex:

```
const add = (a: number, b: number) => {
    return a + b;
}
```

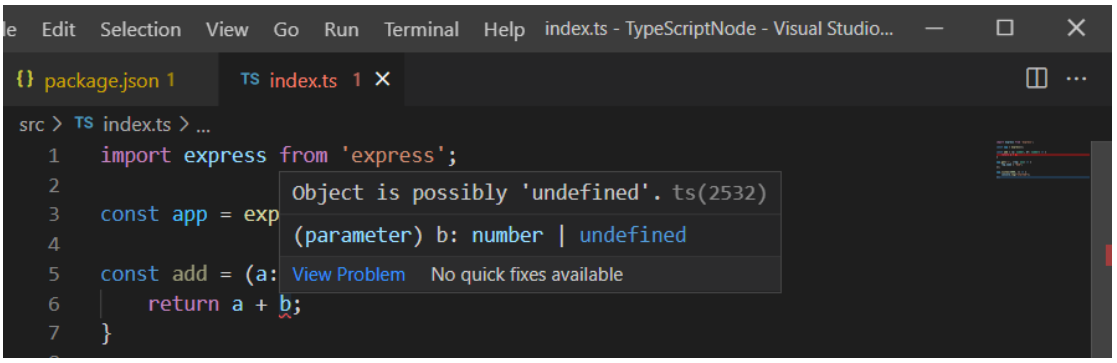
  You can also declare the return type if you want to be explicit, but ts can infer that automatically.
  Ex:

```
const add = (a: number, b: number): number => {
    return a + b;
}
```

- Now, to define a variable as optional, you need to use the ? character.
  Ex:

```
const add = (a: number, b?: number) => {
    return a + b;
}
```

- But now ts is yelling at you because it might be **undefined**, and you can't add a **number** to **undefined**.
  Ex:



- In that case, you need to condition the use of the optional param by whether it's **undefined** or not.
  Now the error should disappear, because ts infers that if **b** equals **true**, it must be defined.
  Ex:

```
const add = (a: number, b?: number) => {
    if (b) {
        return a + b;
```

```
    } else {
        return a;
    }
}
```

- But sometimes ts fails to infer that, and the error doesn't disappear. What then?
  In these cases, you can use the ! character to assert that the variable is definitely not **undefined**.
  Ex:

```
const add = (a: number, b?: number) => {
    return a + b!;
}
```

- Last thing you should know about is **interfaces** and **types**.
    - They are both ways to define **objects**, and **functions** (without implementing them).
    - They are mostly interchangeable - though different people might use them for different uses according to their personal conventions.
    - One possible convention is that **objects** are defined as **interfaces**, and **functions** (and everything else) are defined as **types**.
    - For example, say we want to define an **object** that represents parameters for the **function** from the previous section.
      We'd create an **interface** named "**Params**", and define a single parameter as type **Params**.
      Ex:

```
interface Params {
    a: number,
    b: number
}

const add = (params: Params) => {
    return params.a +  params.b;
}
```

      One reason to do that is that if more **functions** receive this exact set of parameters, we'll be able to just declare a single parameter of type **Params**, and not repeat the declaration of all the actual parameters.

    - Now, we can also define a **function** that represents all functions that receive **Params** as a parameter, and return a **number**.

To do that, we'd create a **type** named "**ParamsMethod**" (or any name that makes sense), and define the parameter as before.
Ex:

```
interface Params {
    a: number,
    b: number
}

type ParamsMethod = (params: Params) => number;
```

One reason to do that is that now we have a **function** "**base type**", and if for example we need to pass as parameter a **function** that receives **Params** as a parameter and returns a **number**, we can just declare a parameter of type **ParamsMethod**, and when we implement the **function** itself, we can define it as of type **ParamsMethod**, so if we use a wrong type somehow, we'll get an error.
Ex: