

---

# Introduction to TypeScript

---

TypeScript is a superset of JavaScript, it is like JS on steroids. It provides syntactic sugar to basic JavaScript which in turn provides end to end safety.

TypeScript compiles down to JavaScript, which means that we can use it wherever you use JavaScript. JS can be used in the front end via a browser, in the back end via Node.js and Deno.

TypeScript uses type inference to give excellent type support without any additional download or IDE configuration.

Using Node.js as a platform, we will setup an environment for TS and then discuss some of its capabilities. We will use several demos and illustrations throughout the bootcamp.

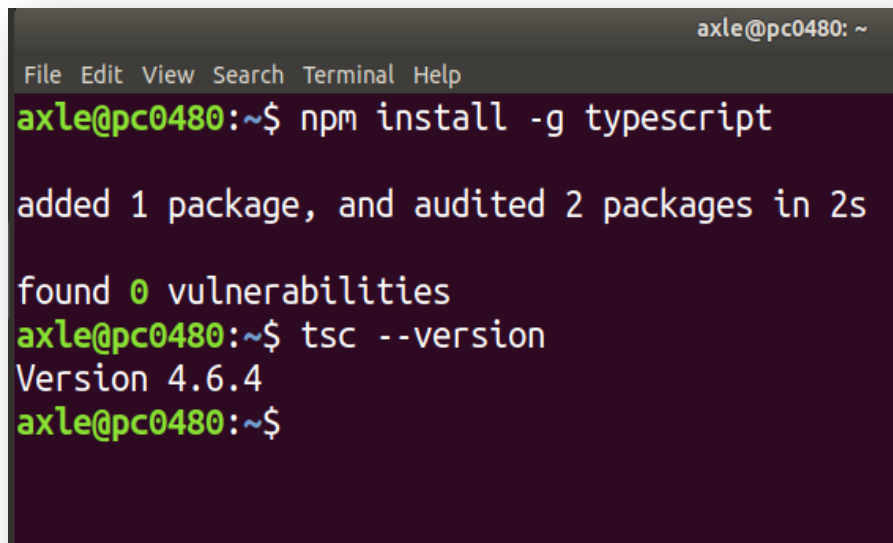
## Part 1 – Environment and IDE

1. Install TypeScript compiler using this command:

```
npm install -g typescript
```

2. Test the install using this command:

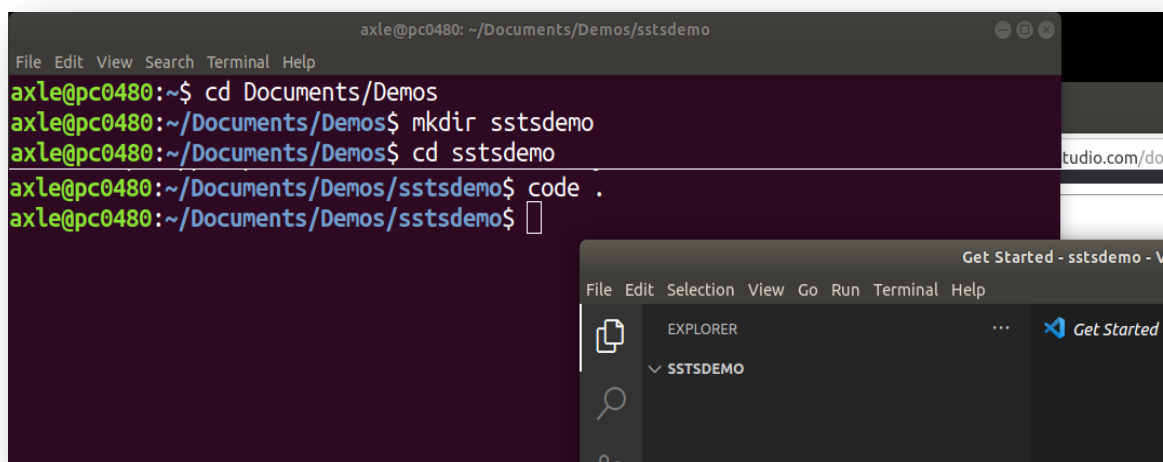
```
tsc -version
```



```
axle@pc0480: ~  
File Edit View Search Terminal Help  
axle@pc0480:~$ npm install -g typescript  
  
added 1 package, and audited 2 packages in 2s  
  
found 0 vulnerabilities  
axle@pc0480:~$ tsc --version  
Version 4.6.4  
axle@pc0480:~$
```

As of this bootcamp, version should be 5.8.3

3. In your OS choose a place where you want to create a working directory, in my case I am using the Documents folder where I have a Demos folder and there I will create a new



```
axle@pc0480: ~/Documents/Demos/sstsdemo  
File Edit View Search Terminal Help  
axle@pc0480:~$ cd Documents/Demos  
axle@pc0480:~/Documents/Demos$ mkdir sstsdemo  
axle@pc0480:~/Documents/Demos$ cd sstsdemo  
axle@pc0480:~/Documents/Demos/sstsdemo$ code .  
axle@pc0480:~/Documents/Demos/sstsdemo$
```

The second screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left shows the file structure with a folder named 'SSTSDemo'. The main editor area is titled 'Get Started - sstsdemo - V' and contains a 'Get Started' button.

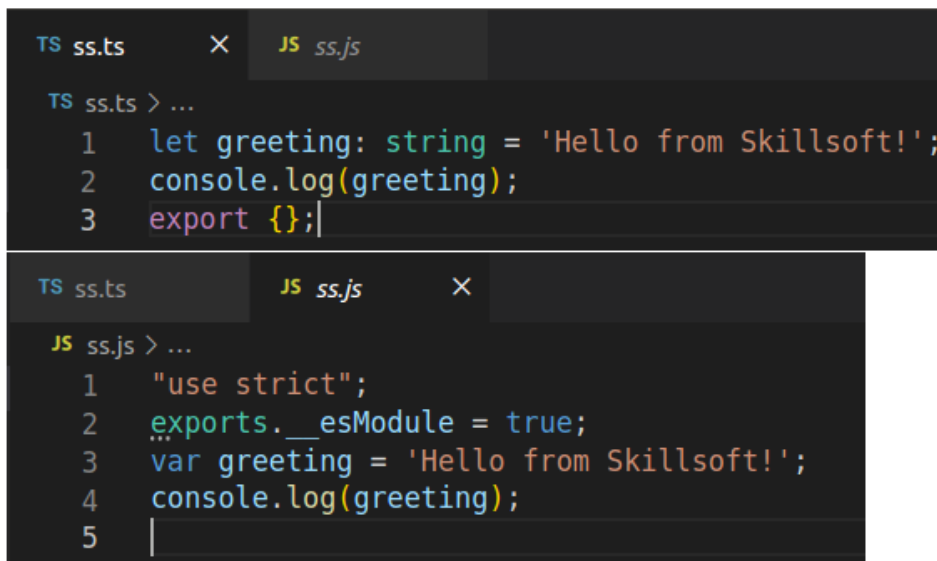
folder called sstsdemo:

Note: the command `code .` will invoke the **VSCode** editor at that location wherever it happens to be (the `.` is the period key on the keyboard)

4. Use the editor to create a new file called ss.ts
5. Once you have a `.ts` file, add the following code so that we can test that everything is working:

```
let greeting: string = 'Hello from Skillsoft!';
console.log(greeting);
export {};
```
6. If you enter the `ls` command at the terminal window, you will see your ss.ts file there. Now if we compile this file to JavaScript we need the **tsc** command, so enter this command:

```
tsc ss.ts
```
7. Now if you run the `ls` command again you will see the JavaScript file sitting there. We can compare both files in VSCode:



```
TS ss.ts  X  JS ss.js

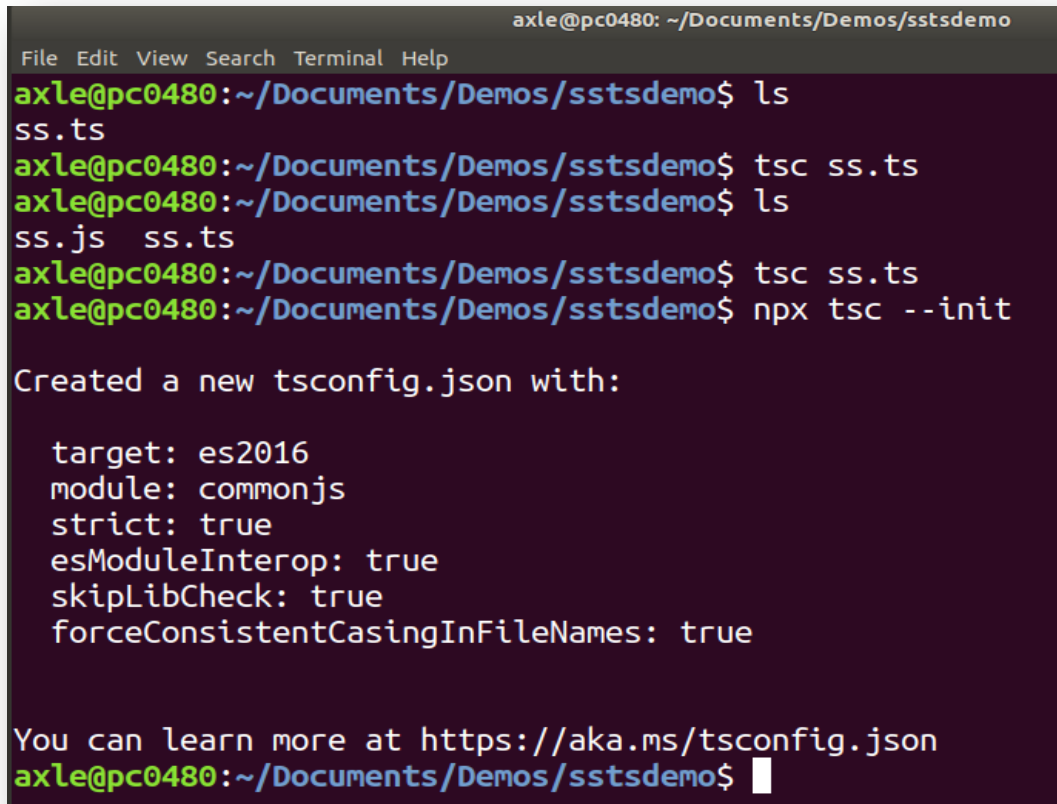
TS ss.ts > ...
1  let greeting: string = 'Hello from Skillsoft!';
2  console.log(greeting);
3  export {};
```

```
TS ss.ts  JS ss.js  X

JS ss.js > ...
1  "use strict";
2  exports.__esModule = true;
3  var greeting = 'Hello from Skillsoft!';
4  console.log(greeting);
5  |
```

Note: we cannot execute the file as yet, JavaScript runs in an environment like a browser or in NodeJS

8. Create a tsconfig.json file in the terminal window, by running the command

A terminal window with a dark background and light-colored text. The title bar at the top reads 'axle@pc0480: ~/Documents/Demos/sstdemo'. The menu bar below it includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows a series of commands and their outputs: 'ls' lists 'ss.ts'; 'tsc ss.ts' runs successfully; 'ls' lists 'ss.js' and 'ss.ts'; 'tsc ss.ts' runs successfully; and 'npx tsc --init' creates a new 'tsconfig.json' file. Below the command, the contents of the file are displayed: 'target: es2016', 'module: commonjs', 'strict: true', 'esModuleInterop: true', 'skipLibCheck: true', and 'forceConsistentCasingInFileNames: true'. A message follows: 'You can learn more at https://aka.ms/tsconfig.json'. The prompt 'axle@pc0480:~/Documents/Demos/sstdemo\$' is visible at the bottom with a cursor.

```
axle@pc0480: ~/Documents/Demos/sstdemo
File Edit View Search Terminal Help
axle@pc0480:~/Documents/Demos/sstdemo$ ls
ss.ts
axle@pc0480:~/Documents/Demos/sstdemo$ tsc ss.ts
axle@pc0480:~/Documents/Demos/sstdemo$ ls
ss.js  ss.ts
axle@pc0480:~/Documents/Demos/sstdemo$ tsc ss.ts
axle@pc0480:~/Documents/Demos/sstdemo$ npx tsc --init

Created a new tsconfig.json with:

  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig.json
axle@pc0480:~/Documents/Demos/sstdemo$
```

`npx tsc -- init` or just create one manually. We will use the command to create this config file and then remove the items we do not want. Note: you could use the terminal window in VSCode.

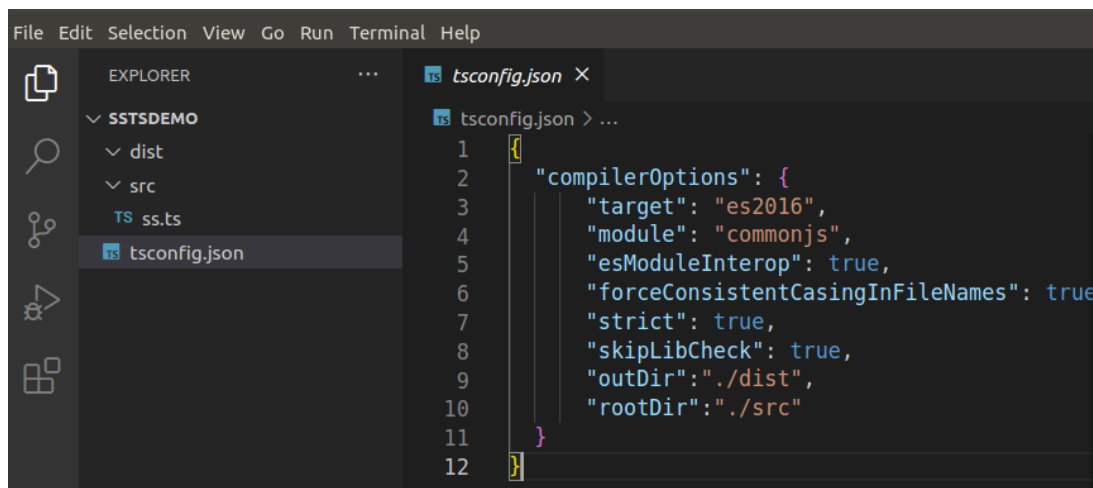
9. Lets set up two directories a src folder and a dist folder as required by the *tsconfig* file. Do this in the **root** of your project.
10. Around line 62 of the tsconfig.json file there is an *outDir* setting, remove the comments and add in an *out* directory:  
`"outDir": "./dist",`
11. We will also configure the input folder, so around line 30 there should be a *rootDir* setting:  
`"rootDir": "./src",`  
Note, you may get an error from the IDE, ignore it for now

12. Here is a sample file:

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true,
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Note: we will be adjusting the *commonjs* setting in a future section

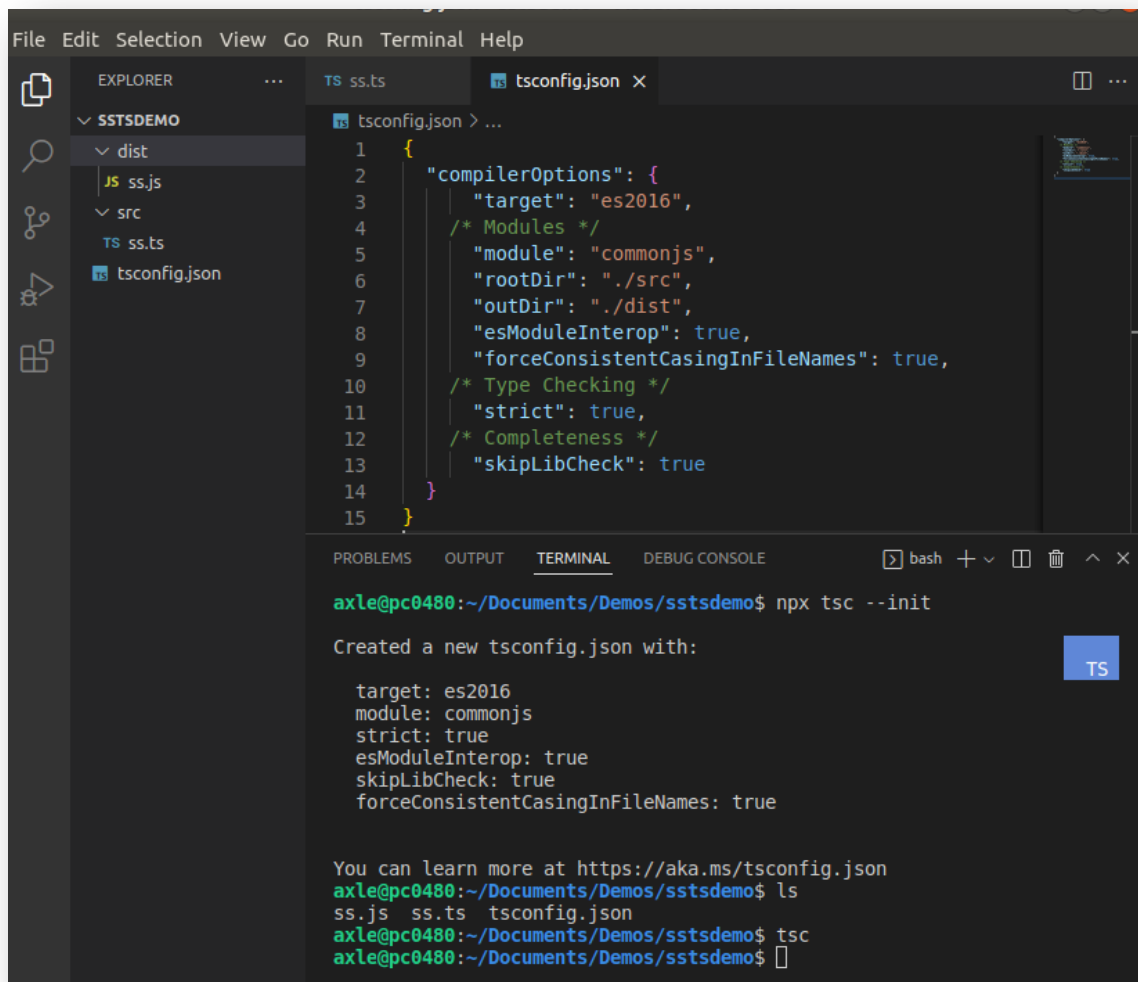
13. this is my setup showing the folder structure and the tsconfig.json file without all the commented out lines:



14. To do a test, move the ss.ts file into the src folder, also delete the ss.js file.

15. Run the command again, so run `tsc` once you delete the original ss.js file you will see that the deleted .js file is replaced in the dist folder once the command is run. NOTE: do not pass the ss.ts file to the command like this `tsc src/ss.ts`

## Introduction to TypeScript



```
File Edit Selection View Go Run Terminal Help

EXPLORER
SSTSDemo
├── dist
│   └── ss.js
├── src
│   ├── ss.ts
│   └── tsconfig.json
└── ...

tsconfig.json > ...
1 {
2   "compilerOptions": {
3     "target": "es2016",
4     /* Modules */
5     "module": "commonjs",
6     "rootDir": "./src",
7     "outDir": "./dist",
8     "esModuleInterop": true,
9     "forceConsistentCasingInFileNames": true,
10    /* Type Checking */
11    "strict": true,
12    /* Completeness */
13    "skipLibCheck": true
14  }
15 }
```

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
axle@pc0480:~/Documents/Demos/sstdemo$ npx tsc --init

Created a new tsconfig.json with:

target: es2016
module: commonjs
strict: true
esModuleInterop: true
skipLibCheck: true
forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig.json
axle@pc0480:~/Documents/Demos/sstdemo$ ls
ss.js  ss.ts  tsconfig.json
axle@pc0480:~/Documents/Demos/sstdemo$ tsc
axle@pc0480:~/Documents/Demos/sstdemo$
```

This is my final setup on a Linux OS

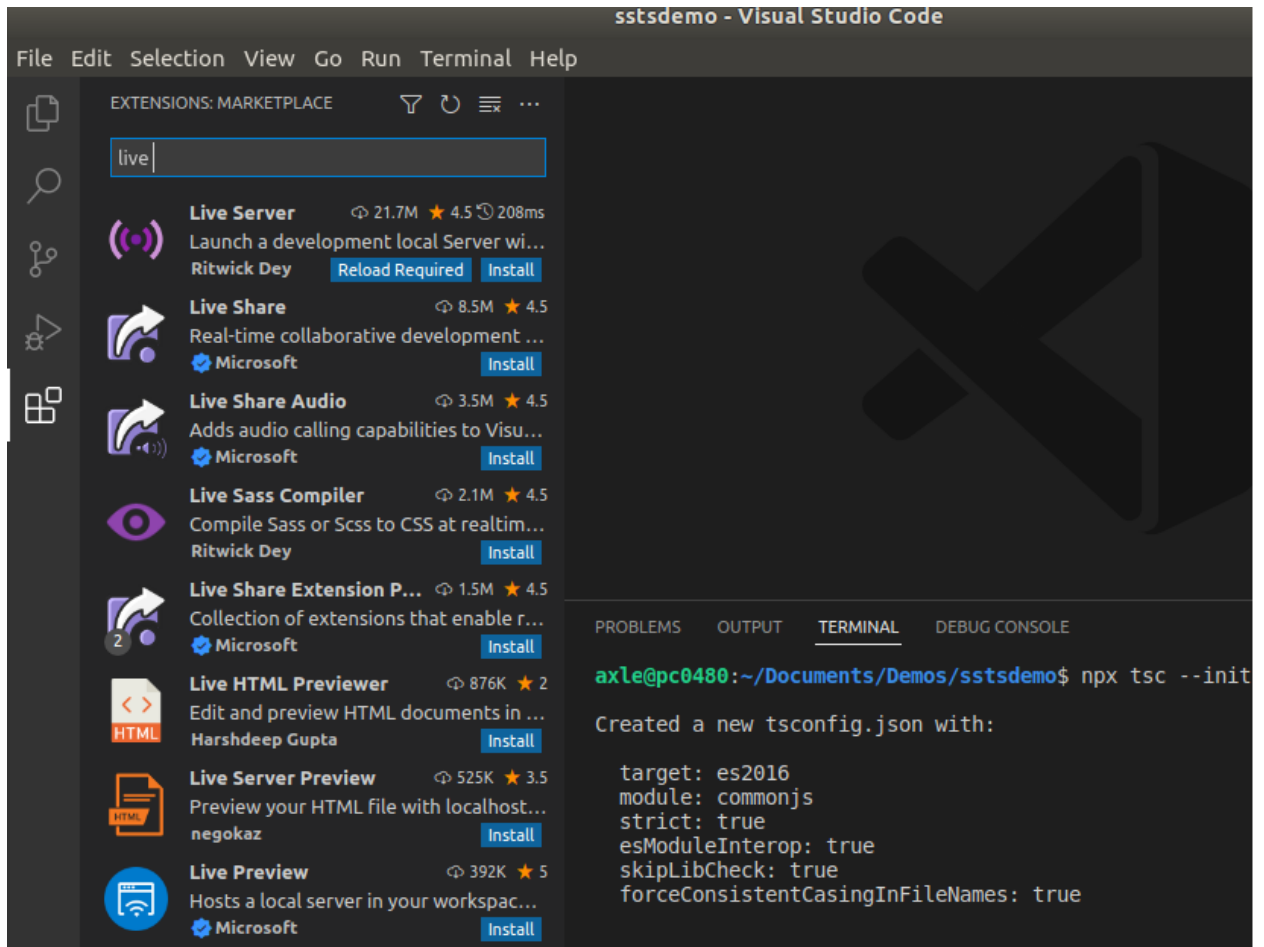
## Part 2 – HTML Setup


1. In the src folder add a new **html** file. You can do this from the IDE, just right click in the src folder and choose the *New File* option. Name the file index.html.
2. Put your cursor inside the index.html file and in the blank editor window start typing *html*, the IDE will respond with three options, choose *html:5* and an HTML template will appear in the file, save it. You can change the *title* if you want to “Learn TS”.
3. Add the following line just before the `<title>` tags:  

```
<script src="index.js"></script>
```
4. Delete the ss.ts and ss.js files from their respective folders and create a new index.ts file in the src folder.
5. If you get an error in the *tsconfig* file, just close the project folder from VSCode and re-open the project folder. You must do this from the IDE, so go to *File -> Close Folder*. Once the IDE closes the project, you may open it again using the shortcut or the *File* menu.

## Introduction to TypeScript

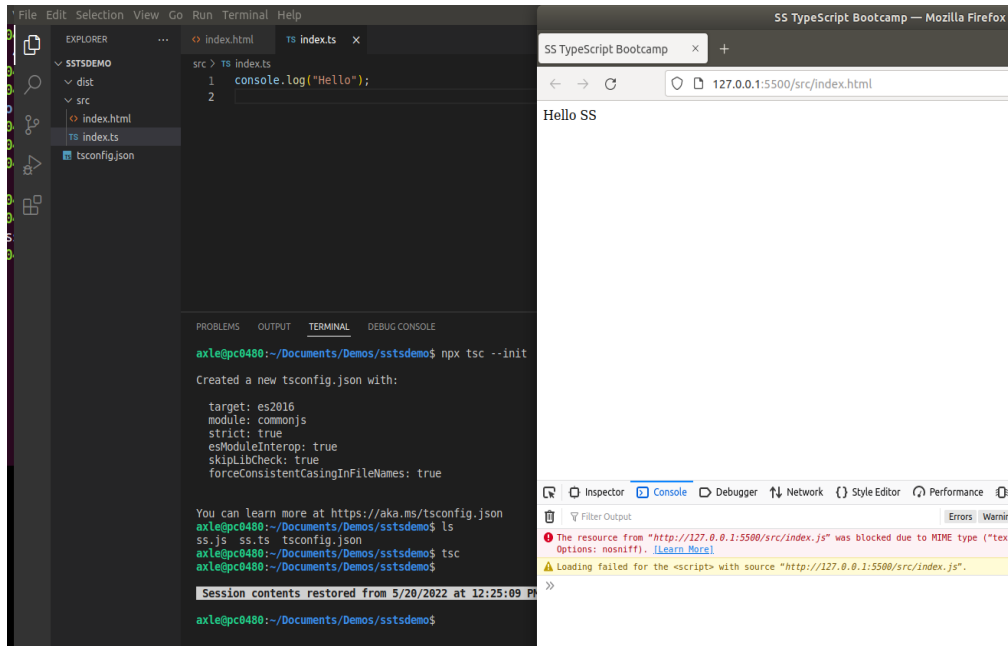
6. In VSCode, click on the **Extensions** icon and in the search bar search for *Live Server*:



7. Choose the server by *Ritwick Dey* and install it using the blue install button. Click on the *Files* icon in the IDE once the server has been installed, it's the  icon. If you are on the Skillsoft sand box, you may already have this configured for you.
8. Once you are absolutely sure that **Live Server** is installed, right click on the index.html file and choose **Open with Live Server**. Your default browser (Firefox in my case) should open showing the blank rendered contents of the index.html file we created earlier. Note, nothing will show right now in the main window, but you should be able to see the *title* of the page in the browser. We set the title in step 2 above.



- If you now position both the IDE and the browser next to each other you will see that the browser responds immediately to any change in the HTML file. For example if we type *Hello* into the `<body>` tags, the browser responds, even without saving the HTML file.
- Open the `index.ts` file in the IDE and add a `console.log()` line like this:  
`console.log("Hello");`

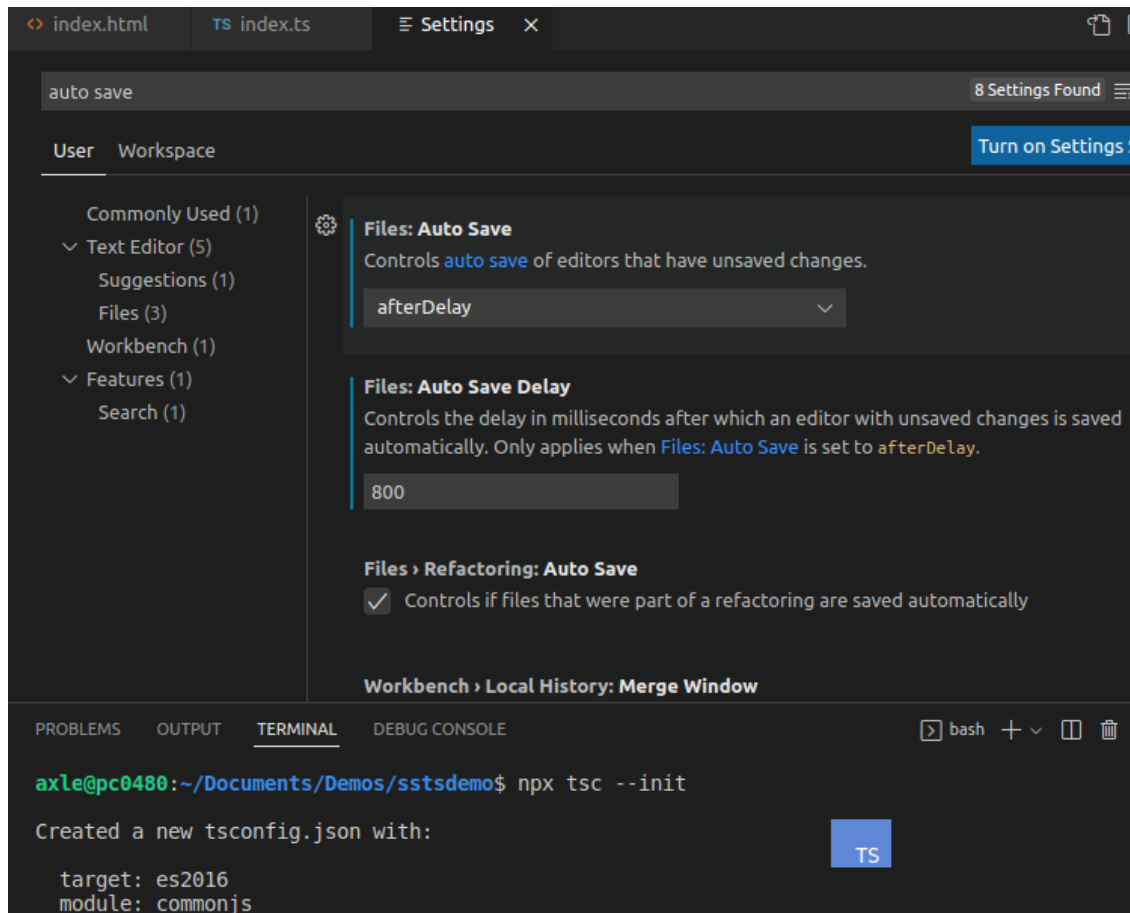


Do not save the file, but now in the browser hit the **F12** key to view the developers tools at the bottom of the browser. Make sure the *Console* tab is opened. There may be an error there but the console log message is not displayed.

- Now back in the IDE, at the bottom left there is a gear icon, click it and choose the Settings option from the pop-up menu.

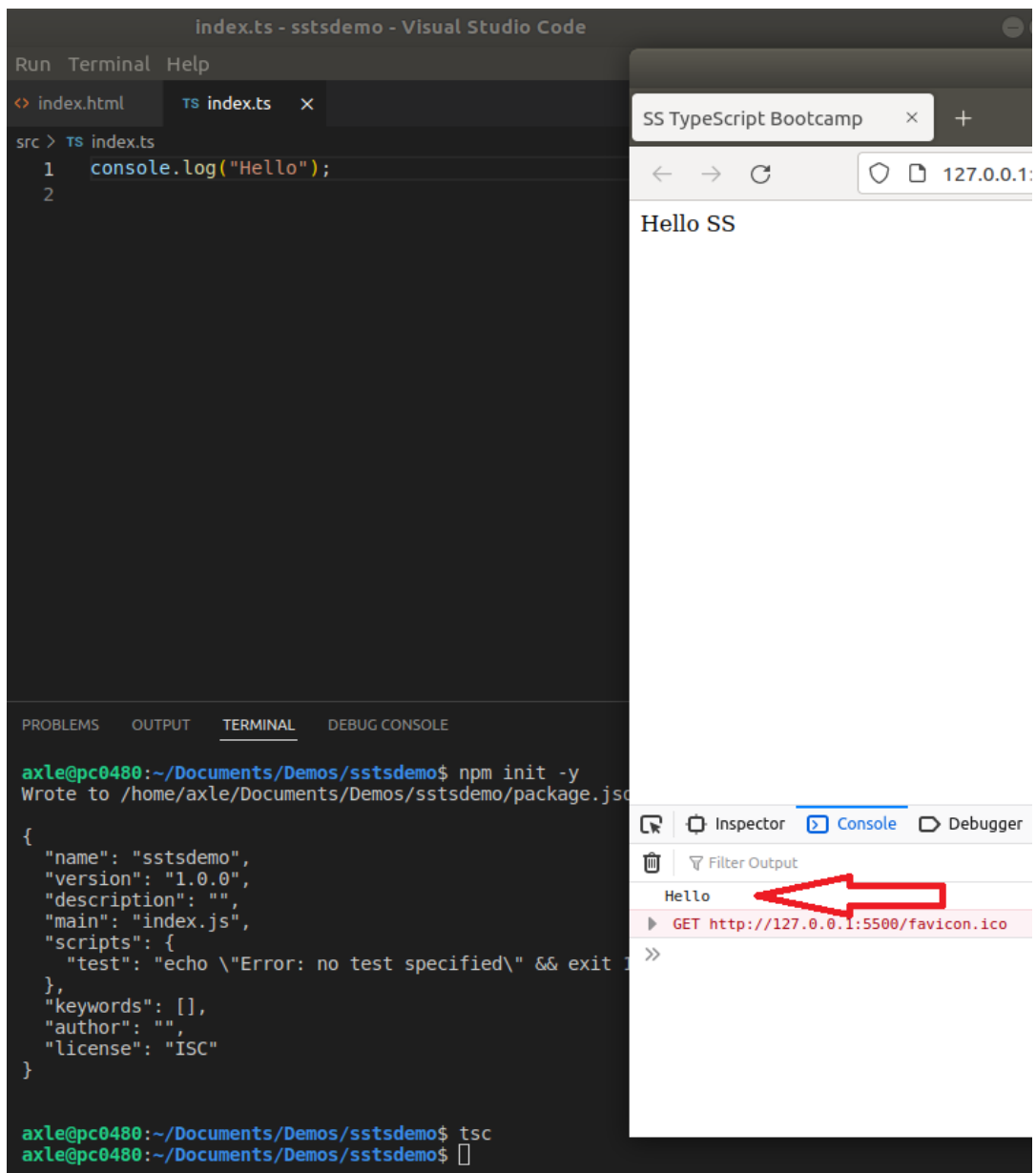
## Introduction to TypeScript

12. In the setting search bar, type in *auto save*, choose *afterDelay* and change the next setting to about 800, which is milliseconds



Close the window, the settings are saved. Note, can be done from “file” menu in VS Code.

13. Now let's setup a *Node.js* project, we will NOT be using Node, but this gives us a package.json file which is like a config file. So in the terminal run:  
`npm init -y`
14. Since we setup the project to look in the dist folder for the index.js file, we need to add the browser path to reflect this, so add the `<script>` tags between the `<head>` tags in the browser setting, make sure your path is pointing to the right file:  
`<script src="../../dist/index.js"></script>`
15. Now run the `tsc` command from the terminal window and a new index.js file should be generated inside of the dist folder and the browser should refresh and now show *Hello* in the *Console* window in the developers area at the bottom.



16. (Optional) The last thing to do in this section is to run the `tsc` command in *watch* mode, in this way anytime we make a change to the `.ts` file, the change will be affected immediately. To do this run the command: `tsc -w`
17. If you did the above, try changing some content that is being logged in the console window and see if the change happens right away. We configured the delay in step 13 above.

## Introduction to TypeScript

The image consists of two screenshots of the Visual Studio Code editor interface, demonstrating the initial setup of a TypeScript project.

**Top Screenshot:** The Explorer sidebar on the left shows a project named "SSTSDemo" with a "dist" folder containing "ss.js" and a "src" folder containing "ss.ts" and "tsconfig.json". The main editor window displays the content of "tsconfig.json":

```
1 {
2   "compilerOptions": {
3     "target": "es2016",
4     /* Modules */
5     "module": "commonjs",
6     "rootDir": "./src",
7     "outDir": "./dist",
8     "esModuleInterop": true,
9     "forceConsistentCasingInFileNames": true,
10    /* Type Checking */
11  }
```

**Bottom Screenshot:** The same project structure is shown. The main editor window displays the completed "tsconfig.json" file:

```
1 {
2   "compilerOptions": {
3     "target": "es2016",
4     /* Modules */
5     "module": "commonjs",
6     "rootDir": "./src",
7     "outDir": "./dist",
8     "esModuleInterop": true,
9     "forceConsistentCasingInFileNames": true,
10    /* Type Checking */
11    "strict": true,
12    /* Completeness */
13    "skipLibCheck": true
14  }
15 }
```

The bottom panel shows the "TERMINAL" tab with the following commands and output:

```
axle@pc0480:~/Documents/Demos/sstsdemo$ npx tsc --init
Created a new tsconfig.json with:

target: es2016
module: commonjs
strict: true
esModuleInterop: true
skipLibCheck: true
forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig.json
axle@pc0480:~/Documents/Demos/sstsdemo$ ls
ss.js  ss.ts  tsconfig.json
axle@pc0480:~/Documents/Demos/sstsdemo$ tsc
axle@pc0480:~/Documents/Demos/sstsdemo$
```

## Part 3 – Working with Types

1. Start this section with this basic code of an employee object:

```
const employee = {
  empName: "Axle",
  dependents: 2,
}
console.log(employee)
```

Just replace the code you have in the index.ts file with what you see here.

2. If you hover over the object's name, so *employee* you will see that TS has correctly inferred that this object is of type string and number:



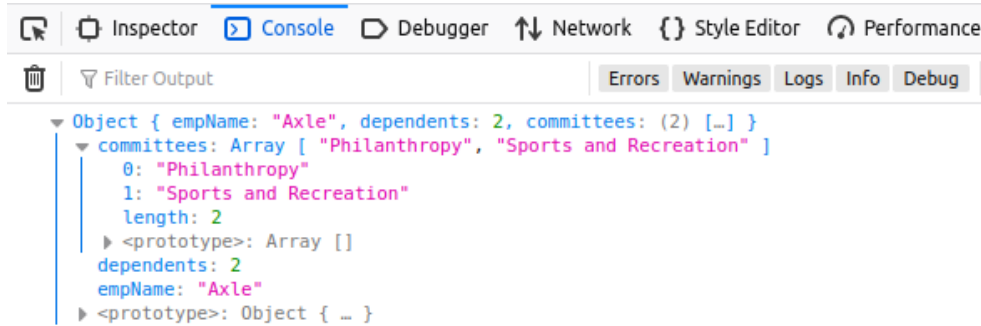
3. Lets add an array to the employee object:

```
const employee = {
  empName: "Axle",
  dependents: 2,
  committees: ["Philanthropy", true]
};
console.log(employee);
```

This is possible in TS, we can have an array of string and Boolean and if you hover over the *committees* key, you will see this inference being made by the TS compiler.

## Introduction to TypeScript

4. The problem with the above code is that we can now attempt to add a string into position 1 (zero based) of the *committees* array, and TS will just accept it:
- ```
employee.committees[1] = "Sports and Recreation";
```



This may or may not be the desired output, that second element is reserved for a Boolean value NOT a string!

5. The solution is to enforce the type checking that TS gives us, it is a feature of TS. In other words, we need a *tuple* and this is the way to declare a tuple:

```
const employee : {  
  empName : string,  
  e: number,  
  committees : [string, boolean]  
} = {  
  empName:"Axle",  
  dependents:2,  
  committees:["Philanthropy", true]  
}  
employee.committees[1] = "Sports and Recreation";//error  
console.log(employee);
```

Normally TS would infer the object, but in this case we want to define to TS what our object should look like. We also want to tell TS that the *committees* array, now **tuple**, is made up of a string and Boolean in that order. This means that if we try to insert a string into position 1, the IDE shows an error. Please note that the *push()* method of arrays and tuples will actually work here, it overrides this check, but that's an advanced topic.

6. TS also supports the *enum* structure:

```
enum weekDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
const employee : {
  empName : string,
  dependents : number,
  committees : [string, boolean],
  payDay : number;
} = {
  empName:"Axle",
  dependents:2,
  committees:["Philanthropy", true],
  payDay : weekDay.FRIDAY
}
```

Notice that in the definition of employee, payday is a number but in the initialization it is a name or string. Once an enum is created it is given a numeric system behind the scenes. We can force a string, see below.

7. We can force the enum to work with strings by simply redeclaring each value:

```
enum weekDay {
  MONDAY = "MONDAY",
  TUESDAY = "TUESDAY",
  WEDNESDAY = "WEDNESDAY",
  THURSDAY = "THURSDAY",
  FRIDAY = "FRIDAY"
};
const employee : {
  empName : string,
  dependents : number,
  committees : [string, boolean],
  payDay : string;
} = {
  ...
```

## Part 4 – Literal Types and Aliases

Continue using the same files from Part 3, just delete everything from the index.ts file. You could of course save the contents of that file if you find it useful, but here in Part 4 we start with a blank .ts file.

1. If you hover the mouse over each of these statements, the first will show string but the second will show **Med**. The second is a *literal* type in TS.

```
let size = "Med";
const SIZE = "Med";
```

2. Literal types are excellent for use with functions. Here is a function that takes two values one string the other number, and it **returns** either a string or a numeric type:

```
function compareWords(
  a : string, b : number) : "" | 0 {
  if(...)
    return "";
  else
    return 0;
}
```

3. Literal types can be returned from functions.

```
function compareWords(
  first : string, second : string) : "first is earlier"
| "same" | "first is later" {
  if(first < second)
    return "first is earlier";
  if(first > second) return "first is later";
  return "same";
}
console.log(compareWords("Axle", "Axle"));
```

This function will take two strings, compare them and return one of three and **only one** of three values. Note: this function can be re-written like this and can be found on the official docs for TS:

```
function compare(a: string, b: string): -1 | 0 | 1 {
  return a === b ? 0 : a > b ? 1 : -1;
}
```



- To demonstrate another example using union types, enter this function into the IDE and run the program:

```
function generateId(useNumeric: boolean): string | number
{
    if (useNumeric) {
        return Math.floor(Math.random() * 100);
    } else {
        return "Employee ID:"
        + Math.random()
        .toString(36)
        .substring(6)
    }
};
```

`Math.random()` generates a random decimal between 0 and 1 which is then converted to base-36. The substring part removes the first 5 characters which gets rid of the zero

- We could test the above code like this:

```
        .substring(6)
    }
};
console.log(generateId(true));
console.log(generateId(false));
```

You will receive different results due to randomness but the first `log()` will show a numeric value and the second a string like `abc123`

- If you tried to do math from the true-based `log()`, it will work in the console window but show an error in VS Code. This is because TS cannot guarantee you will receive a numeric value:

```
console.log(generateId(true) + 1);
console.log(generateId(false));
```

- This will guarantee a numeric value:

```
const id = generateId(true);
console.log(typeof id === "number" ? id + 1 : id);
console.log(generateId(false));
```

8. **Aliases** work just like types, we could declare one or more union primitive types using an alias, then replace the individual union with the alias:

```
type returnable = string | number;
function generateId(useNumeric: boolean): returnable {
  if (useNumeric) {
    return Math.floor(Math.random() * 100);...
```

```
8   .toString(36)
9   .substring(6)]
10 }
11 };
12 //      function generateId(useNumeric: boolean): returnable
13 const id = generateId(true);
14 console.log(typeof id === "number" ? id + 1 : id);
15 console.log(generateId(false));
```

If you did use an alias, then anywhere that type is used, if you hover over the type, you will see the alias identifier.

## Part 5 –Function Type, Callback and Higher Order Functions

Continue using the same files from Part 4, just delete everything from the index.ts file. You could of course save the contents of that file if you find it useful, but here in Part 5 we start with a blank .ts file.

1. In the code below, we start with an array and then define a function:

```
const numbers = [1, 2, 3, 4, 5];
type Operation = (num: number) => number;
```

In some languages this second line is known as a function signature

- Now we define another function called `iCalc()`. This function takes two parameters, the first is an array of numbers and the second is a function:

```
function applyOperation(
  numbers: number[],
  operation: Operation
): number[] {
  return numbers.map(operation);
};
```

- We could now have a function that matches the signature from #1

```
function double(x: number): number {
  return x * 2;
}
```

The `double()` function takes a numbers and returns a number, just like `Operation`.

- It means then that we could assign the concrete function to our `applyOperation()` function (without calling it):

```
let doubledNumbers = applyOperation(numbers, double);
```

Here we call `applyOperation()` and pass it exactly what it needs, an array and a function. This will work since the `double()` function matches the signature that `applyOperation()` takes.

- If we now call `iCalc` and pass it the required parameters, we should see a numeric output

```
console.log("Doubled:", doubledNumbers);
```

- Note, we could eliminate one line by putting the function signature directly into the function that uses it:

```
function applyOperation(
  numbers: number[],
  operation: (num: number) => number
): number[] {
  return numbers.map(operation);
}
```

7. (Optional) You could actually reduce the entire file to just four lines (or shorter) but it's a bit more confusing:

```
const numbers = [1, 2, 3, 4, 5];
const applyOperation = (nums: number[], op: (n: number) => number) => nums.map(op);
const double = (x: number) => x * 2;
console.log("Doubled:", applyOperation(numbers, double));
```

## Part 6 – Chaining and Null Coalescing

Continue using the same files from Part 5, just delete everything from the index.ts file. You could of course save the contents of that file if you find it useful, but here in Part 6 we start with a blank .ts file.

1. If you visit the public, community-based **jsonplaceholder** site (<https://jsonplaceholder.typicode.com/posts>) and try to access the posts data, you see data that is wrapped in this format:

```
userId : 1
id : 1
title : ""
body : ""
```

2. We could now define a type based on this data:

```
type post = {
  userId : number;
  postId : number;
  title : string;
  body : string;
};
```

3. Lets now create an actual post for demonstration and log the details or part of the details

```
const myPost : post = {
  userId : 100,
  postId : 3842,
  title : "TypeScript Rocks",
  body : {topic:"Decorators", explanation:"Adds
functionality to functions"}
};
//
console.log(myPost.body);
```

Note: this could be in a loop

Also change the **post** -> **body** to { topic : string, explanation : string }

4. This is the index.ts file so far:

```
type post = {
  userId : number;
  postId : number;
  title : string;
  body : { topic : string, explanation : string };
};
const myPost : post = {
  userId : 100,
  postId : 3842,
  title : "TypeScript Rocks",
  body : {topic:"Decorators", explanation:"Adds
functionality to functions"}
};
console.log(myPost.body);
```

5. But there could be a problem, this is data that we do not control, what if there is no *topic* or *explanation* being returned? One JS solution is to do something like this:

```
if(myPost.body && myPost.body.topic)
  console.log(myPost.body.topic);
```

With this code, if *body* exists, then JS will continue on to check *topic*. However if *body* does NOT exist, *topic* is **never** checked. This code works in TS and in JS and is called short circuiting.

6. You can test the above code by passing an empty string in the **body.topic** variable:

```
    title : string;
    body : { topic : any, explanation : string };
...
    } title : "TypeScript Rocks",
    body : {
        topic: "",
        explanation: "Adds functionality to functions"
    }
```

Change the topic type to be *any* and add an empty string to *topic* in the body section.

This will not print anything for *body.topic*.

7. But TS has a better, shorter solution. Its called **Optional Chaining**.

```
    body : {topic:"Decorators", explanation:"Adds
functionality to functions"}
};
//
if (myPost.body?.topic)
    console.log(myPost.body.topic);
```

Much shorter, but we still do not get anything being printed for *myPost.body.topic*.

This code uses TypeScript's optional chaining operator (*?.*) to safely access a nested property (*topic* in this case).

If **myPost.body** is null or undefined, the entire expression will short-circuit and return *undefined* instead of throwing a runtime error. However, if **myPost.body** exists and has a value, it will continue the property access and return the value.

8. What about if you want to store the data you are getting from an outside source:

```
let posts : string[] = [];
posts.push(myPost.body.topic);
console.log(posts);
```

Here we create an array of posts and attempt to push each topic into the array. In this case we get an empty string if your **topic** is still an empty string.

9. Since we expect that the topic could be *null* or *undefined*. Well we could add a default value or a fail-safe value like this:

```
let posts : string[] = [];
posts.push(myPost.body.topic || "No Topic");
```

Here if we do not get a value for topic, we simply add a **default value** to the array at that location. On the other hand, if **topic** did have a value, we get that value and not the No Topic value.

10. Lets do a test, passs an empty string:

```
    title : "TypeScript Rocks",
    body : {topic:"", explanation:"Adds functionality to
functions"}
};
//
let posts : string[] = [];
posts.push(myPost.body.topic || "No Topic");
console.log(posts);
```

It prints "No Topic" because topic is now treated as *falsey*. Of course this may be exactly what we want.

11. There are times however when we only want to eliminate **undefined** or **null values**. If that is the case we must use the **nullish coalescing operator** in TS

```
    title : "TypeScript Rocks",
    body : {topic:"", explanation:"Adds functionality to
functions"}
};
//
let posts : string[] = [];
posts.push(myPost.body.topic ?? "No Topic");
console.log(posts);
```

Now it will print an empty string in the space where this element is loaded into the array. In other words, it wont print "No Topic".

12. If you now explicitly set *topic* as *null* or *undefined*, you get the “No Topic” default string being stored:

```
    title : "TypeScript Rocks",
    body : {topic:undefined, explanation:"Adds
functionality to functions"}
};
//
let posts : string[] = [];
posts.push(myPost.body.topic ?? "No Topic");
console.log(posts);
```

If you tried this, you would need to change the definition of *body* in *type post* to something like this: `body : {topic:any, explanation:string };`

13. Here is the entire code for this section:

```
type post = {
  userId : number;
  postId : number;
  title : string;
  body : { topic : any, explanation : string };
};
//
const myPost : post = {
  userId : 100,
  postId : 3842,
  title : "TypeScript Rocks",
  body : {
    topic:undefined,
    explanation:"Adds functionality to functions"
  }
};
//
//if(myPost.body?.topic)
  //console.log(myPost.body.topic);

let posts : string[] = [];
posts.push(myPost.body.topic ?? "No Topic");
console.log(posts);
```





---

## Appendix A : Favicon Error

---

To get rid of the favicon error add the following to the <head> tag of the browser:

```
<link rel="shortcut icon" href="#">
```

This will also work:

```
<link rel="icon" href="data:,">
```

---

# Appendix B : Topics

---

Part 1 – Environment and IDE

Part 2 – HTML Setup

Part 3 – Working with Types

Part 4 – Literal Types and Aliases

Part 5 –Function Type and Callback Functions

Part 6 – Chaining and Null Coalescing

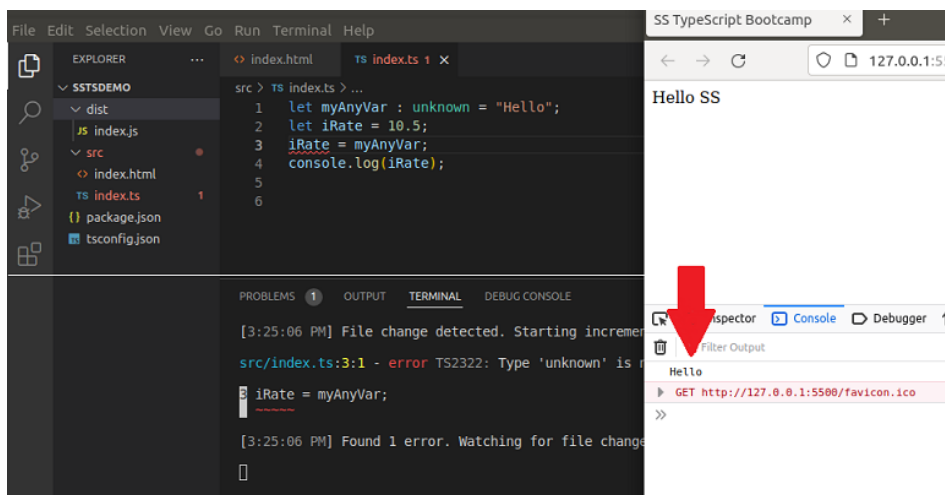
## Appendix C : Optional Config Settings

### 1. `removeComments` : `true`

This setting will allow you to enter *comments* in the .tsc file but once compiled into a .js file, the comments will be removed

### 2. `noEmitOnError` : `true`

In order to show what this setting does, lets introduce an error:



The code has an error, we are trying to assign an unknown variable type to a number type. The IDE shows the error but the code still compiled, notice the red arrow in the browser. If we change or add the `noEmitOnError` and set it to `true`, this compilation will NOT happen. The default is `false`.

### 3. You can set the following three at all at once:

```
noUnusedLocals : false
noUnusedParameters: false
noImplicitReturns : false
```

With this in place if you have functions in your code and you create variables but don't use them, the IDE will complain. Also if you declare functions with parameter(s) and don't supply them it will complain. If you create functions that does now always return a value it will also complain. This happens when you have `if` statements but don't provide a `false` part.

---

## Appendix D : Uncaught Reference Error

---

If you get this error in the browser:

*Uncaught ReferenceError: exports is not defined*

Make sure your `"module": "commonjs"` line in the `tsconfig.js` file is commented out. Also make sure your target is **es2016** or above.

Finally remove the line `export {}` from your `index.ts` file if you have it.

---

## Appendix E : Function Parameter in JS

---

```
"use strict";
const y = 9;
const x = function (n1, n2) {
  return n1 + n2;
};
//
function addem(z, y, x){
  return x(z, y);
}
let result = addem(1,2,x)
console.log(result);
```

Part 4 #3

```
function compareWords(
  first : string, second : string
) : "first is earlier" | "same" | "first is later" {
  const result = first.localeCompare(second);
  return result < 0 ? "first is earlier" : result > 0 ? "first is later" : "same";
};
```