|  |
| --- |
| Close-up image showing the leaf-sides of two oversized books side-by-side on a bookshelf, with additional books in soft focus background |
| Getting Started with Typescript for SharePoint Framework Development |
| |  |  |  | | --- | --- | --- | | Priyaranjan KS |  |  | |

# About the Author

**About the Author**

Priyaranjan KS is a Senior SharePoint Consultant, who is engaged in architecting, designing and developing solutions in SharePoint and Office 365.He has been working with SharePoint over the past 10 years and has worked on SharePoint 2007 through SharePoint 2019.

He is a Certified Scrum Master, as well as a Microsoft Certified Solutions Developer (SharePoint Apps). He is a Microsoft MVP since 2018.

**Target Audience**

This book is intended to help you get started with Typescript and how we can use them with SharePoint Framework. The target audience is required to have a basic understanding of JavaScript so that the samples will be easier to follow. New samples will be updated on a regular basis.



Priyaranjan K S

(Microsoft MVP)

# Course Details

Contents

[About the Author 2](#_Toc35793848)

[Course Details 3](#_Toc35793849)

[Setup TypeScript Development Environment 5](#_Toc35793850)

[Install Visual Studio Code Editor 5](#_Toc35793851)

[Install Node.js 5](#_Toc35793852)

[Install Typescript 5](#_Toc35793853)

[Create a Project in TypeScript 6](#_Toc35793854)

[Create a new folder 6](#_Toc35793855)

[Create the typescript file 7](#_Toc35793856)

[Create Html File 7](#_Toc35793857)

[Transpile TS to JS 8](#_Toc35793858)

[Test the Project 8](#_Toc35793859)

[Debug the Application 9](#_Toc35793860)

[Creating Folder Structure 9](#_Toc35793861)

[Package.JSON 10](#_Toc35793862)

[HTTP Server 11](#_Toc35793863)

[Typescript Project Files 12](#_Toc35793864)

[Compiler Options 12](#_Toc35793865)

[Variable Declaration 16](#_Toc35793866)

[Var Declarations 16](#_Toc35793867)

[Let declaration 17](#_Toc35793868)

[Block Scoping 18](#_Toc35793869)

[const declarations 18](#_Toc35793870)

[Basic Types 19](#_Toc35793871)

[Union Types 23](#_Toc35793872)

[Type Assertions 24](#_Toc35793873)

[Type Annotations 24](#_Toc35793874)

[Type Annotations with Functions 25](#_Toc35793875)

[Arrow Functions 26](#_Toc35793876)

[Interfaces 26](#_Toc35793877)

[Inheritance of Interface 27](#_Toc35793878)

[Multiple Interface Inheritance 28](#_Toc35793879)

[Classes 29](#_Toc35793880)

[Class Inheritance 30](#_Toc35793881)

[Getting Started with SharePoint Framework Development 31](#_Toc35793882)

[Set up SharePoint Framework Development Environment 31](#_Toc35793883)

[Install Node JS 32](#_Toc35793884)

[Install Yeoman and Gulp 36](#_Toc35793885)

[Install Yeoman SharePoint Generator 37](#_Toc35793886)

[Code Editor 37](#_Toc35793887)

[Additional Tools for Development and Debugging 40](#_Toc35793888)

[Postman 42](#_Toc35793889)

[Getting Started with SharePoint Framework Development using TypeScript 43](#_Toc35793890)

[Create the First Hello World Client Web part 43](#_Toc35793891)

[Create the Web part project 43](#_Toc35793892)

[Test the web part 46](#_Toc35793893)

[SharePoint Workbench 47](#_Toc35793894)

[Edit the web part 49](#_Toc35793895)

[Add the web part to SharePoint 51](#_Toc35793896)

[Create SharePoint Framework Client Web Part to Retrieve and Display List Items 52](#_Toc35793897)

[Create the Web part Project 53](#_Toc35793898)

[Test the Web part locally 55](#_Toc35793899)

[Edit the web part 56](#_Toc35793900)

[Define List Model 56](#_Toc35793901)

[Create Mock HTTPClient to test data locally 57](#_Toc35793902)

[Retrieve SharePoint List Items 58](#_Toc35793903)

[Render the SharePoint List Items from Employee List 59](#_Toc35793904)

[TS File Contents 60](#_Toc35793905)

[Mock HTTP Client Content 63](#_Toc35793906)

[Test the Web part in local SharePoint Workbench 63](#_Toc35793907)

[Test the Web part in SharePoint Online 64](#_Toc35793908)

[Summary 66](#_Toc35793909)

# Setup TypeScript Development Environment

## Install Visual Studio Code Editor

We need to install VS (Visual Studio) Code editor for typescript development. You can download VS Code Editor here (https://code.visualstudio.com/download ).

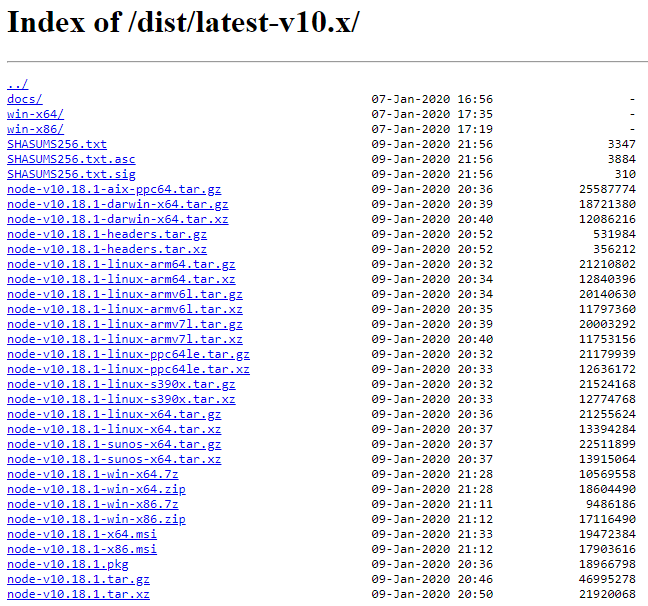
## Install Node.js

Next we will first install NPM which comes with Node JS. Node JS contains Node Package manager which is required to install Type Script

We can install Node JS from the official site, <https://nodejs.org/en/> . However if we are installing Node JS for Typescript development using SharePoint Framework , we will have to be careful as SPFx supports Node v10. We can get the previous version from here

<https://nodejs.org/en/download/releases/>

Select version 10.x . This will take us to the download location of the file. Spot the 64 bit msi version which will let us install v10 to the system



## Install Typescript

There are multiple ways to install typescript, one of the recommended ways is to visit the official web site <https://www.typescriptlang.org/> . Clicking on the Download button will take us to the page where we can see various editor options that we can use with TS. Clicking on them would ideally add the latest version of TS to the editor. Say for instance if we click on VS, it will add the latest version of TS to VS.

However if we want to use it with VS Code, clicking on it will take us to the VS Code, download page which will let us download the VS Code which will already contain the latest version on TS.

More over we can also see how to use TS development with editors by clicking on the editor in the And More section

**Installing the TypeScript compiler**

We will be working with visual Studio code in this course for TS development. Visual Studio Code **includes TypeScript language support but does not include the TypeScript compiler**, tsc. You will need to install the TypeScript compiler either globally or in your workspace to transpile TypeScript source code to JavaScript (tsc HelloWorld.ts).

The easiest way to install TypeScript is through npm, the Node.js Package Manager.

**Show how to install Node**

Now that we have npm installed, we can install TypeScript globally (-g) on your computer by running the below command as shown in the TS website:

npm install -g typescript

You can test your install by checking the version.

tsc –version

# Create a Project in TypeScript

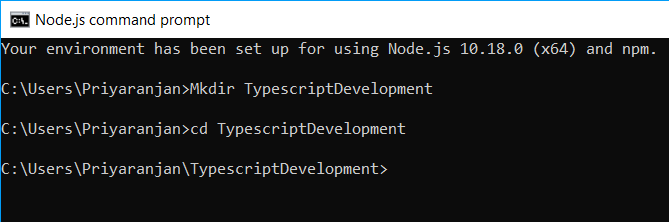
From the command line select Node JS command prompt

## Create a new folder

As the first step, we need to create a folder where we will save all our project files. I would be creating a new folder called *TypescriptDevelopment* using the below command where we will have the parent folder created which stores our project.

mkdir TypescriptDevelopment

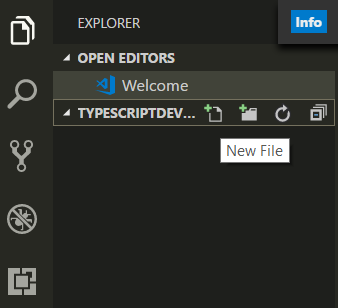
Lets navigate inside using cd command.



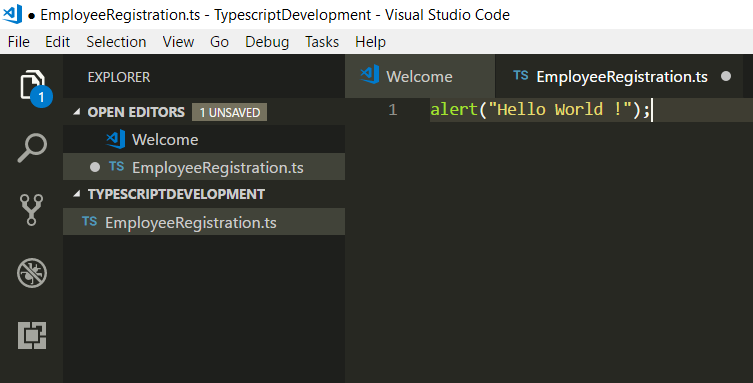
Once we are inside the working directory, run the command code . which will open up the visual studio code editor for us which we will be using to create our Employee Registration app from the scratch.

## Create the typescript file

From the visual studio code, Select New File option which will let us create a file with a specific extension



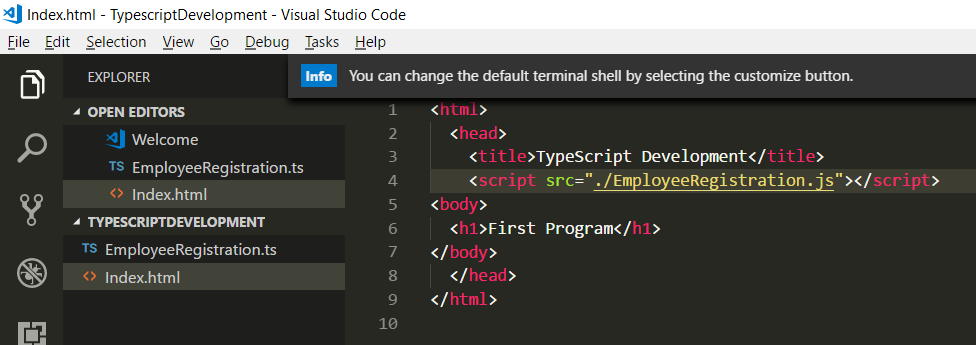
In our case we are going to add a .ts file which will contain the logic or code which will be later converted to Javascript by the compiler. For sample demo, lets use the alert statement to output Hello World !



## Create Html File

Lets click on the New File icon and add a file with .html extension as well.Lets name it as Index.html We will add the below code within the file.

1. <html>
2. <head>
3. <title>TypeScript Development</title>
4. <script src="./EmployeeRegistration.js"></script>
5. <body>
6. <h1>First Program</h1>
7. </body>
8. </head>
9. </html>



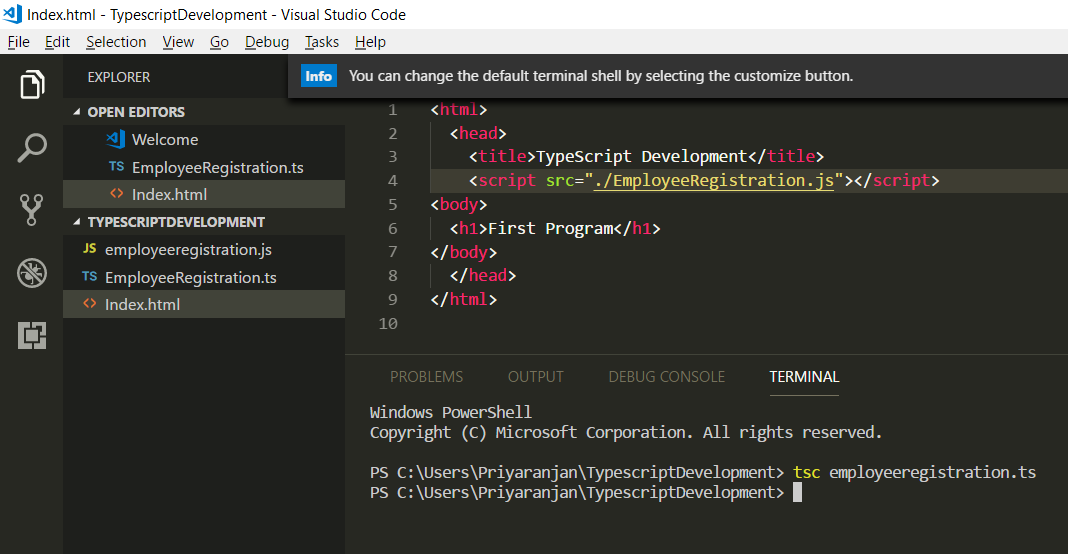
If we look at the code, we can see that we have referenced a JS file with the same name as the TS file that we created previously. The reason is , browsers don’t support Typescript files. Hence we will convert the TS file to JS file using Typescript transpiler

## Transpile TS to JS

So as to Transile the Typescript file and generate the equivalent JS file we will head over the integrated terminal option in the View Option of Visual Studio. Run the command

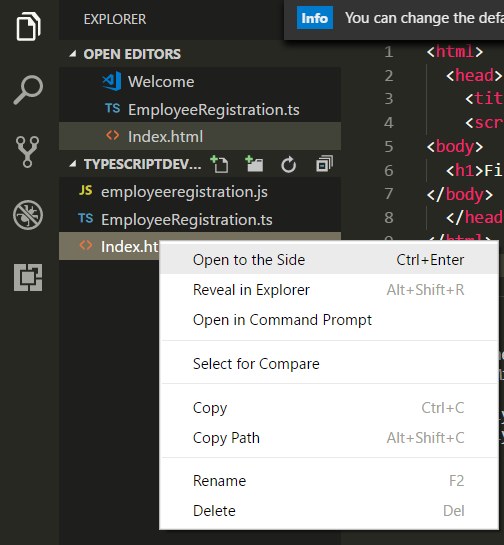
Tsc EmployeeRegistration.ts

This will create the equivalent JS file as shown below which we have referred in the HTML file.

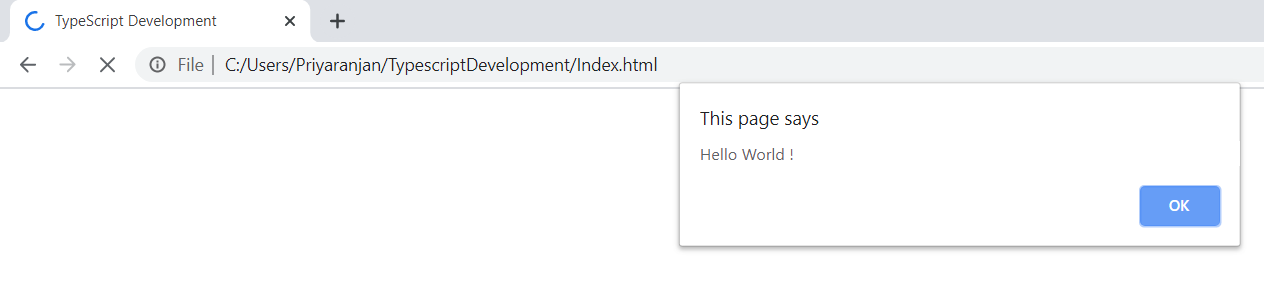


## Test the Project

Now lets run the index.html file. Copy the file path of the index file by going to visual studio code and Select the Copy Path option by right clicking the index file.

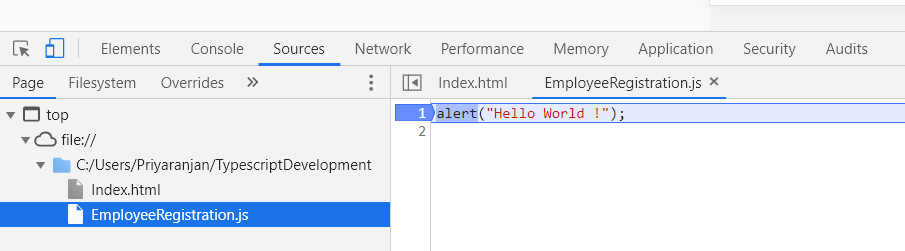


Open the browser and input the path and enter. We can see the Hello World Pop up from the TS application.



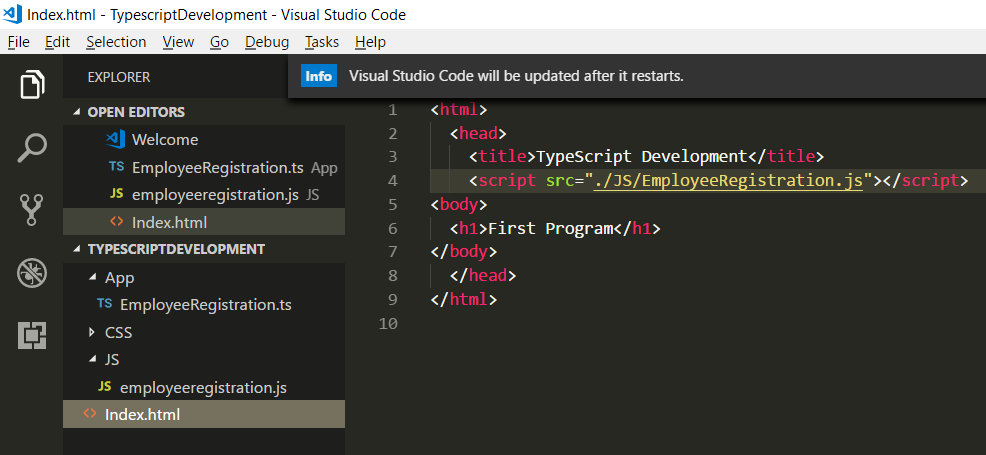
## Debug the Application

If you want to add breakpoints in between code execution to check on some variables and test the code flow, we can open up the browser inspection tool and add breakpoints at the required code statement by going to the Sources tab. On a refresh the code execution would pause at the break point and we can do the inspections.



## Creating Folder Structure

Lets create a basic project structure to isolate the files in their respective folders. We will be creating a JS , CSS and App Folder which will house the Transpiled JS file, any CSS files and the main TS file respectively as shown below.



Ensure to update the JS file path in the index.html to reflect the change in project structure. Currently the TS file is within the App folder. However if we run tsc command to transpile the JS file, it will create the JS file within the App folder. We will see how to direct the JS file to be created directly in the JS Folder later.

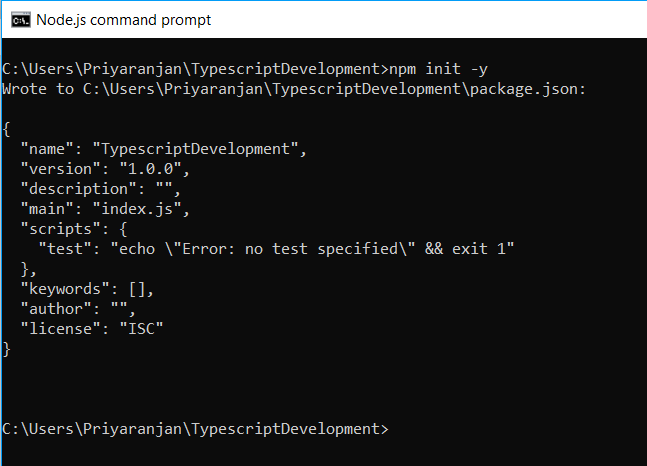
## Package.JSON

This is an NPM file which stores information about the NPM packages that we will be using in the project. In case we plan to publish the package online, there are other json values that governs the publishing process like Name,Version, License etc. However in this course we will be mostly concerned about the devdependency json key which we will talk about in a while.All the key value pairs in the JSON is defined in the below link which will give you a better idea of what each means.

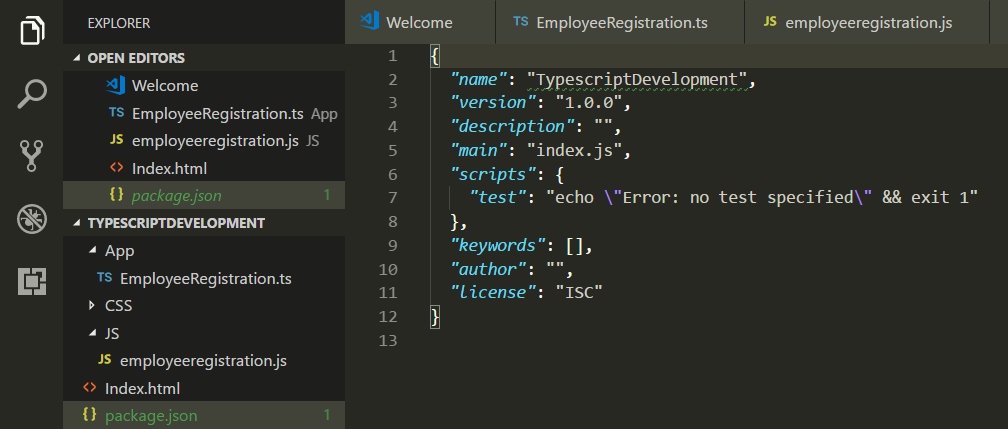
<https://docs.npmjs.com/files/package.json>

In order to create a Package.JSON for the project we can run the below command and retain the default value . In case you want to specify some custom value that we would like to use in the project, you can omit the -y parameter.

Npm -init -y



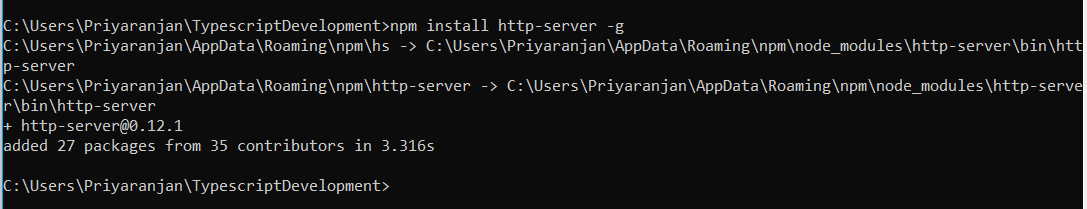
Now lets head over to the VS Code and see the newly created package.json



## HTTP Server

In the previous section of setting up the project file, we tested the solution by getting the path of the HTML file by right clicking the Index.html file. But instead of that we can use the HTTP Server npm package to serve up you project file. It’s a very handy web server that can be used to test the development works. We can install the npm package as shown below :

npm install http-server -g



We will also define the start script for the server within the json file within the scripts key by using start as the script key and the npm command http-server as its value. This indicates that running the start script in the command line will run the command http-server .

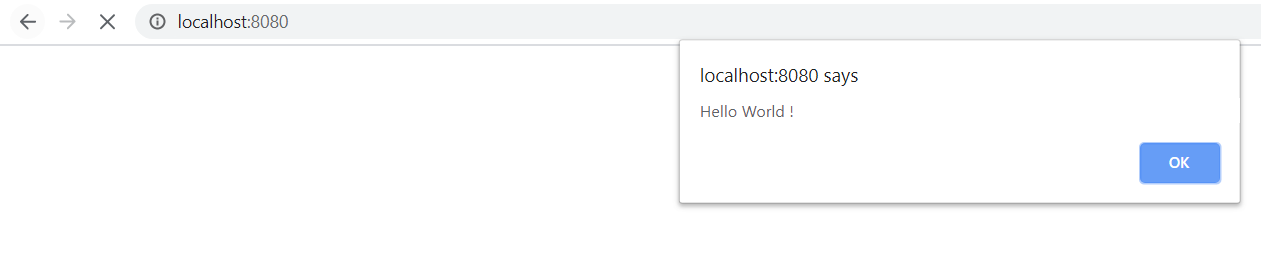
The “scripts” property is a dictionary containing script commands that are run at various times in the lifecycle of your package. The key is the lifecycle event, and the value is the command to run at that point.



Now that we have made the changes, lets test the server with our Hello World App by running npm start in the integrated terminal. This has started the server and its



Now you can visit http://localhost:8080 to view your server.



<https://www.npmjs.com/package/http-server>

# Typescript Project Files

These are plain simple text files that has the name tsconfig.json. The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

It also helps to store the compiler options which otherwise would have to be specified while running the compiler. In addition to it TSConfig can be used to specify which files has to be included or excluded from compilation using include/exclude keys. A tsconfig.json file is permitted to be completely empty, which compiles all files included by default (as described above) with the default compiler options.

<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

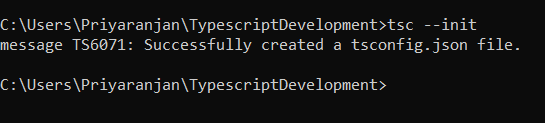
## Compiler Options

TSConfig.json file can be used to govern how the compiler compiles the TS Files by specifying the properties as Key Value Pair within the file.

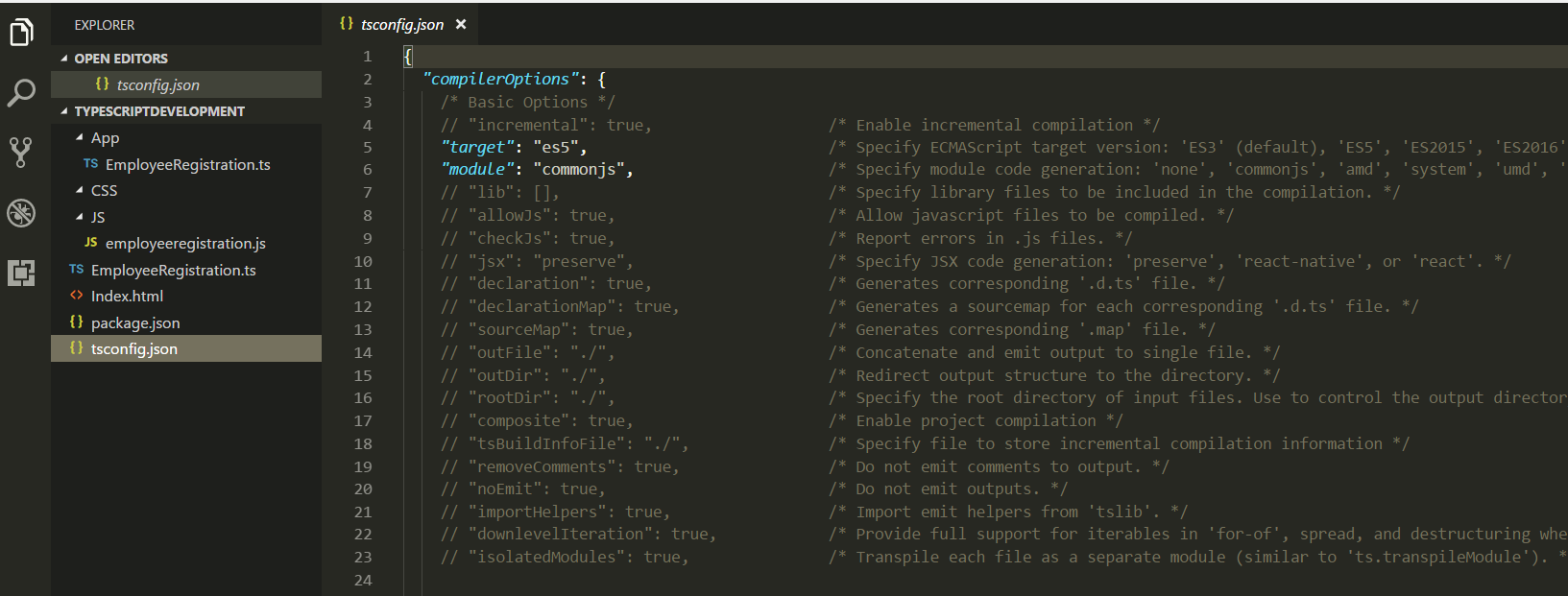


We do not have a TSConfig in the project yet. Lets go ahead and create one by running the below code.

tsc –init



Now if we head over to the project, we can see that the TS Config has been created with some default key value pairs.



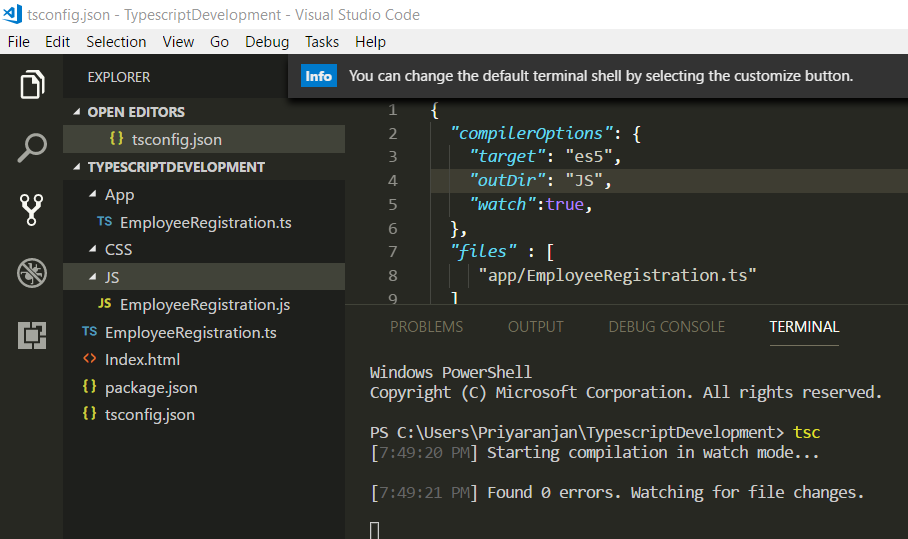
Lets delete the contents and start with a minimalistic configuration for easier understanding. We will go with the below Configuration settings. As we can see the configurations are present as an array within the compilerOptions Key. The "compilerOptions" property can be omitted, in which case the compiler’s defaults are used.

Few of the properties that we have mentioned are

* Target : It specifies to the version of Javscript to which it is being compiled. If we don’t specify any value it will by default be compiled to ES3. Here we have specified ES5 which is a widely supporte version in most of the browsers.
* outDir : It specifies the folder to which the compiled Javsascript file will be saved.In case the folder is not present, it will be created.
* watch : It leaves the compiler running in the background. Whenever the TS file is saved it compiles the JS files automatically and we can see the changes in the JS files in realtime
* files : this property specifies which files should be compiled. It is an array of file names which has to be compiled to create the output JS files. In our case, we are compiling only the app.ts file within the app folder.

1. "compilerOptions": {
2. "target": "es5",
3. "outDir": "JS",
4. "watch":true,
5. },
6. "files" : [
7. "app/EmployeeRegistration.ts"
8. ]

Lets go ahead and delete the JS folder in the current project structure so as to test if the compiler option will create an output directory and place the compiled JS File there.



Running tsc will start the compilation and we can see that the JS folder has been created and the compiled js file with the same name as TS file has been created inside it.

Nows lets take a look at the watch property in action using the below added code :

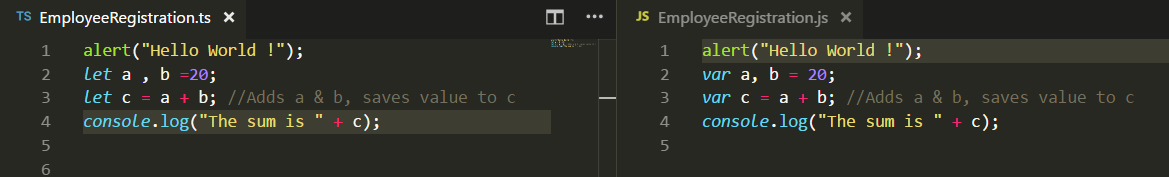
*let* a , b =20;

*let* c = a + b; //Adds a & b, saves value to c

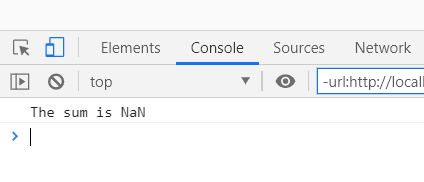
*console*.log("The sum is " + c);

From the file tab in VS Code, I have enabled Auto save so that the code gets saved intermittently.

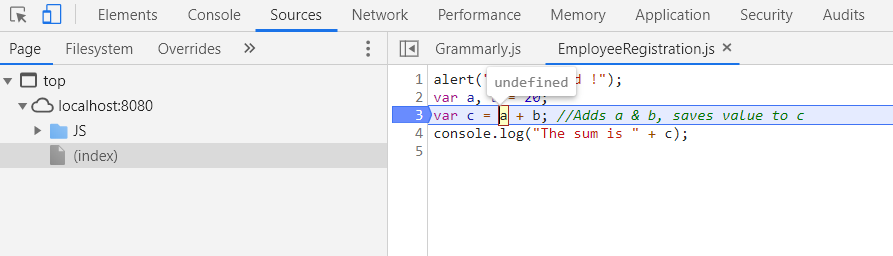
Now as soon as I add the above code to the TS file, it compiles and add the corresponding lines of code in JS file as the Watch is enabled.



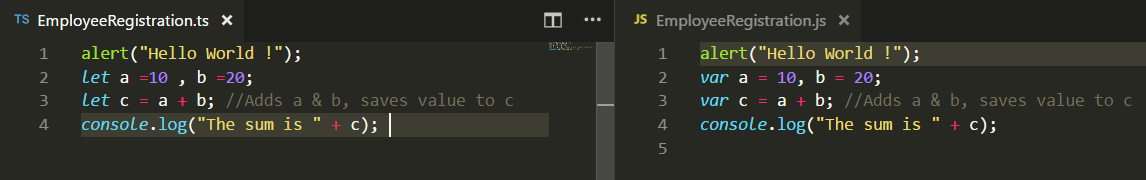
Now lets start the Server by running npm start and see the results in the console in the browser.



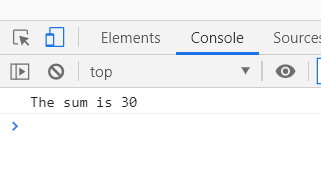
We can see that the result is coming up as NaN. Lets try to debug the code by adding a breakpoint to the JS File in the source tab.



On rerunning it hits the breakpoint and we can see that the variable is declared but not assigned a value which results in the NaN when the calculation happens. Lets go back to the editor and assign a value, run the compiler once again and restart the server.



This time the console has outputted the value correctly.



Thus we saw how we can configure the compiler option and utilize watch to see the JS compilation in real time .

# Variable Declaration

In this chapter we will look at how we can do variable declarations using

* Var
* Let
* Const

let and const are two relatively new types of variable declarations in JavaScript. let is similar to var in some respects, but allows users to avoid some of the common mistake that users run into in JavaScript. const is an augmentation of let in that it prevents re-assignment to a variable.

With TypeScript being a superset of JavaScript, the language naturally supports let and const. Here we’ll elaborate more on these new declarations and why they’re preferable to var.

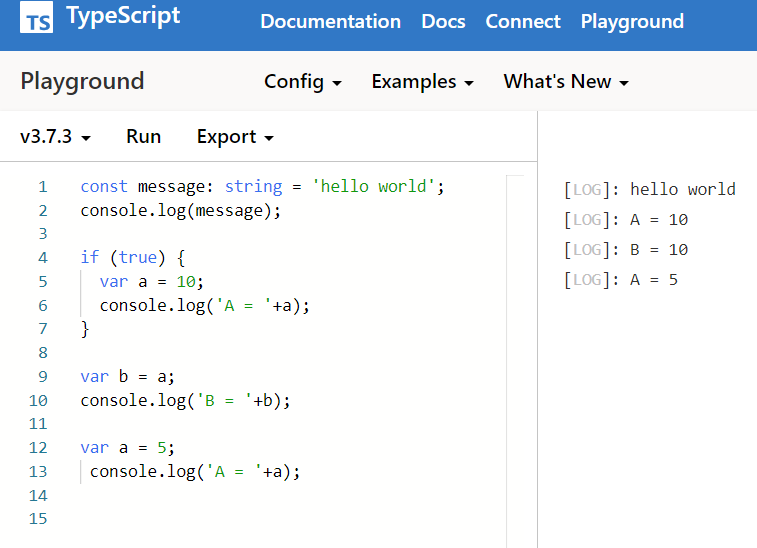
## Var Declarations

We used to traditionally declare variables in Javascript using var key word as :

var a = 50;

which declares the variable a with the number value of 50.

However var comes with the ability to redeclare the same variable multiple times within the function scope which can cause unintentional overwriting of the variable value. Lets head over to the [Typescriptlang Playground](https://www.typescriptlang.org/v2/en/play) and test this out.



1. const message: string = 'hello world';
2. console.log(message);
3. if (true) {
4. var a = 10;
5. console.log('A = '+a);
6. }
7. var b = a;
8. console.log('B = '+b);
9. var a = 5;
10. console.log('A = '+a);

Here we can see that the variable declared within the If block is available outside as well and we are able to set it to variable b without any issues. It also lets us redeclare the variables without any errors for a second time. However, if we do this at 2 different points in a function without knowing that the variable has already been defined with some other value, we are in for some serious logic issues.

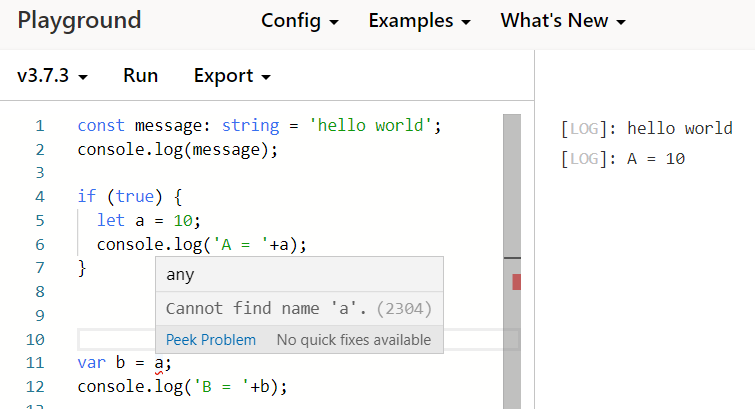
## Let declaration

By now we have figured out that var adds some unintentional code issue, which is precisely why let statements were introduced. Apart from the keyword used, let statements are written the same way var statements are.

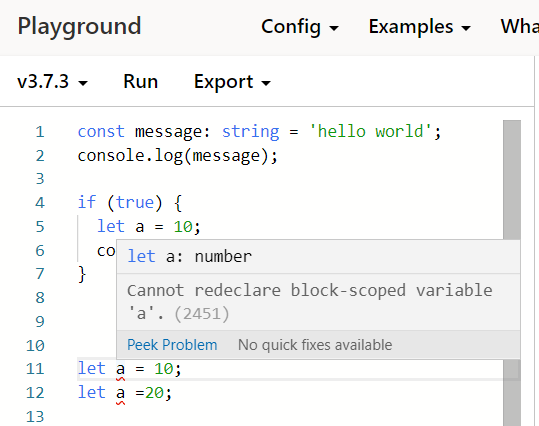
Let a = 50 declares variable a with value 50

### Block Scoping

When a variable is declared using let, it uses what some call lexical-scoping or block-scoping. Unlike variables declared with var whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block. We use the same code as above but we can see that a red squiggle underscore that says a is not found.



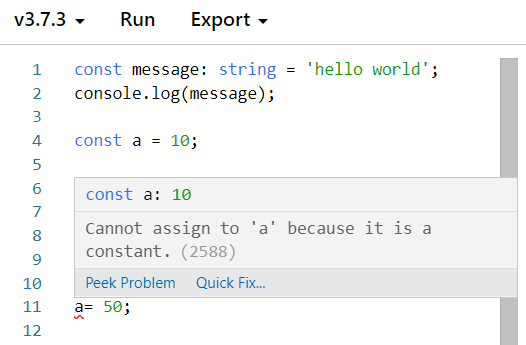
Same way I try to redeclare variable a twice and it throws the error that we cannot redeclare block scoped variable. Thus it provides more native way of reducing code errors imparted due to human negligence.



## const declarations

Just like let, const declarations are another way of declaring variables. However once declared, we cannot change its value.

const a = 10;



They have the same scoping rules as Let just that they cannot be reassigned a value.

# Basic Types

So as to work with the various data types just as in Javsascript, Typescript also has the similar datatypes which are :

* Boolean
* Number
* String
* Array
* Tuple
* Enum
* Any
* Void
* Null and Undefined
* Never
* Object

#### Boolean :

It’s the most basic data type which either accepts True/False value and the declaration goes like :

let flag: boolean = true;

#### **Type Annotation and Type Inference**

In the above section we have used annotations to describe what kind of datatype the flag variable holds. Annotations are specified by placing the semi column just after the variable name followed by the data type.

let flag: boolean = true;

Here the flag is annotated with the Boolean data type and value true is assigned to it.

However Typescript compiler is intelligent enough to understand and infer the datatype even if we do not use Type Annotation.

Let flag = true will be inferred by the compiler that flag is of type Boolean and value is true. If we try to assign a string or number to flag again, it will throw a compile time error.

#### Number

As in JavaScript, all numbers in TypeScript are floating point values. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals.

Variables can be declared as :

1. let decimal: number = 123456789;
2. let hex: number = 75BCD15;
3. let binary: number = 0b1010;
4. let octal: number = 0o744;

#### String

String data type accepts textual data. As in Javascript, we can use Double Quotes or Single Quotes to surround the text data. The declaration syntax goes like :

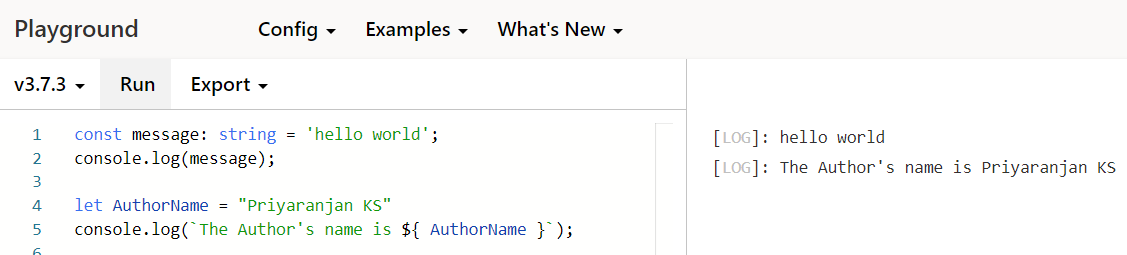
let message: string = "Hello World !";

#### Template Strings

We can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

1. let AuthorName = "Priyaranjan KS"
2. console.log(`The Author's name is ${ AuthorName }`);

Running this in the TypescriptLang Playground would look like below :



#### Array

Just like any other language, TS allows us to use the datatype Array to work with a collection of values. We can define array in two ways :

In the first, we use the type of the elements followed by [] to denote an array of that element type:

let collection: number[] = [1, 2, 3];

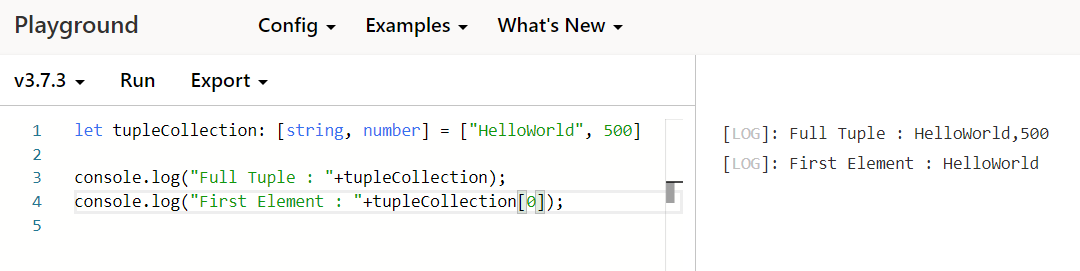
The second way uses a generic array type, Array<elemType>:

let list: Array<number> = [1, 2, 3];

#### Tuple

Tuple types allow one to express an array with a fixed number of elements whose types are known, but need not be the same. For example, we may want to represent a value as a pair of a string and a number which we can do as below:

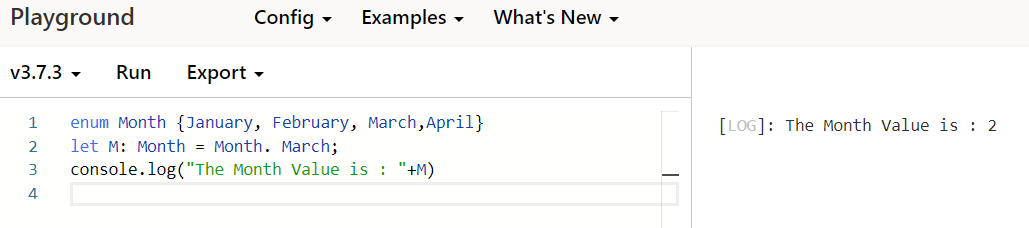
1. let tupleCollection: [string, number] = ["HelloWorld", 500]
3. console.log("Full Tuple : "+tupleCollection);
4. console.log("First Element : "+tupleCollection[0]);



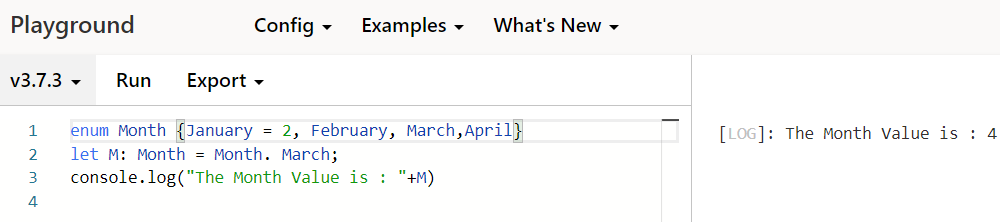
#### Enum

An additional datatype to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

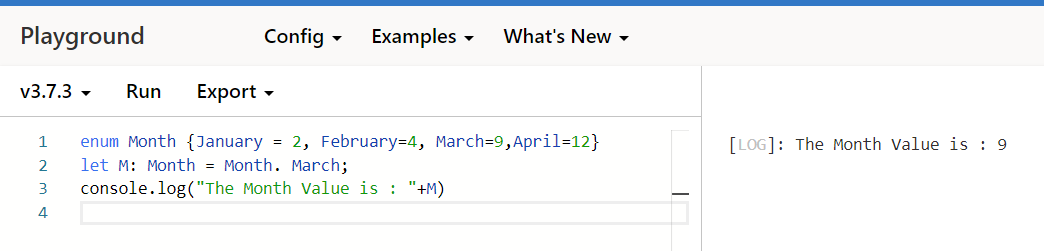
1. enum Month {January, February, March,April}
2. let M: Month = Month. March;
3. console.log("The Month Value is : "+M)



Here the Month enum has month names collection which by default starts value from 0. Hence January will have the value 0 and subsequent elements will have the incremented value. However, if we want the value to start from a different integer, we can specify that as well.



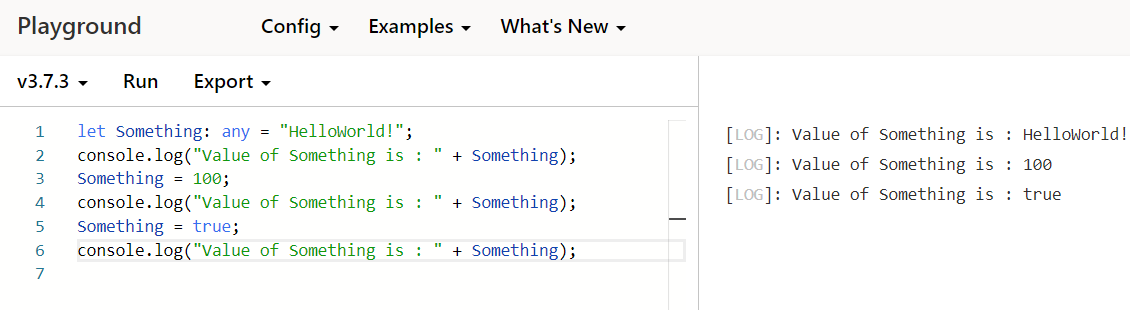
Or if we want to specify the value for each element, that is also possible.



#### Any

We may have to define the type of variables that we do not know when we are writing an application. The values for these variables may be assigned dynamically during run time as a result of some code execution or it may be coming from a third party library. We can use any datatype as :

1. let Something: any = "HelloWorld!";
2. console.log("Value of Something is : " + Something);
3. Something = 100;
4. console.log("Value of Something is : " + Something);
5. Something = true;
6. console.log("Value of Something is : " + Something);



Thus we can assign any values to the variable if it is defined with any datatype and during execution we can assign any values without generating a compile error.

The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

1. Let Collection: any[] = [“January”, 100, false];

#### Void

Void data type is used to indicate the absence of having any type at all. We usually use this as the return type of functions that do not return a value:

1. function ShowMessage(): void {
2. alert("Hello World !");
3. }

Assigning void to a variable is not programatically useful as you can only assign null or undefined to them.

#### Null and Undefined

Typescript also provides us with 2 other types Null and Undefined which can be assigned to variables to show that they hold any values.

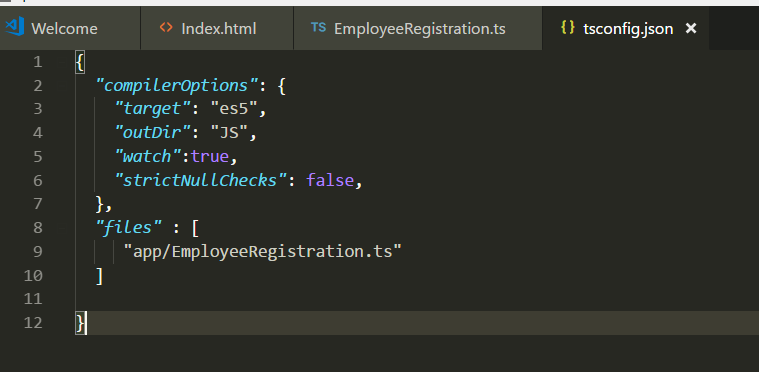
1. let UndefinedVar: undefined = undefined;
2. let NullVar: null = null;

when using the --strictNullChecks flag, null and undefined are only assignable to any type and Undefined/Null Types.Else we can even assign them with other datatypes like string. However as a best coding practise, its adviced to use –strictNullChecks.

##### –strictNullChecks

By default null and undefined are assignable to other data type variables if strict null check is not enabled. Hwever once we have enabled this compiler option, we can the null and undefined values to only those variables that are declares with any type or the null/undefined type.

Assigning null/undefined to any other types will cause compile time errors. We can enable strictNullCheck by changing the property value in the tsconfig.json.



Once it is turned to true, the below code compilation will happen accordingly :

1. let a : number =10
2. a = null //Compilation error
3. let b : null = null //Compiles OK

#### Never

The never data type is used to represent values that will never occur.

## Union Types

There can be a situation where we would expect either a string or a number to be saved to a variable. In such situation TypeScript allows us to use more than one data type for a variable or a function parameter. Let see how to use union types with

##### Variable as Union Type :

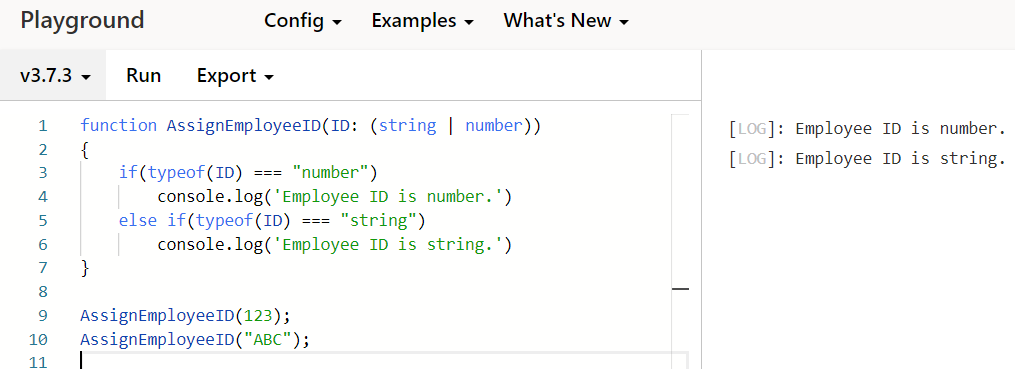
1. let EmployeeID: string | number;
2. EmployeeID = 1;
3. EmployeeID = "Emp1";
4. EmployeeID = False; // Compiler Error

Here the variable EmployeeID can accept either a number value or a string value as we have declared it using Union Type string | number. The vertical bar indicates the union of the possible data types . However any other data type value assigned to it will generate compile time error .

##### Function Parameter as Union Type :

The function parameter can also be of union type,as shown below :

1. function AssignEmployeeID(ID: (string | number))
2. {
3. if(typeof(ID) === "number")
4. console.log('Employee ID is number.')
5. else if(typeof(ID) === "string")
6. console.log('Employee ID is string.')
7. }
8. AssignEmployeeID(123);
9. AssignEmployeeID("ABC");



Here we can see that the ID parameter in the function is assigned a union type so that it can accept either a string or number as the input parameter to the function. Any other data type being passed will generate a compile time error.

# Type Assertions

At times the developer would be in a position to determine what the data type for a variable would be in case of a dynamic assignment scenario. Type assertions are a way to tell the compiler that we know what the data type is going to be, hence we will be type casting that to the desired data type.

We can do type assertion using two syntaxes :

* Using Angle Bracket and specifying the data type inside it

1. let info: any = "Hello Word !";
2. let infoLength: number = (<string> info).length;

Here we are asserting the info variable as string and getting the length of the string.

* Using as Syntax

1. let info: any = "Hello Word !";
2. let infoLength: number = ( info as string).length;

## Type Annotations

As Javascript is not a typed language we wont be able to specify the type of a variable as string or number. This bring in more confusions as we develop the code and start writing more functions. However with Typescript we can can specify the type of variable or parameters by using <**: Type**> after the variable or parameter.This annotation helps the compiler to understand the type of variable or parameter that is being used.

A simple example of type annotation with variable is :

var Score: number = 100; //This tells the compiler that score is a number variable

### Type Annotations with Functions

In plane Javascript the function can be written as

1. function Greetings(salutation, name)
2. {
3. console.log(salutation+” ”+ name);
4. }
5. Greetings(“Hello”, "Priyaranjan");

Here the salutation and name parameter in the Greetings function can take any type and just based on the signature we cannot tell what type the function returns. However reading the code within the function we can understand that it returns a string type. This is not an intuitive way of programming.

To make things intuitive and easy for the compiler to do type checking we can include Type Annotations within the function and the above function can be modified as :

1. function Greetings(salutation : string , name ? : string) : string
2. {
3. return(salutation+' '+ name);
4. }
5. Greetings('Hello', "Priyaranjan");



Here we have annotated the parameters with their types as well as annotated the return type of the function so that by just looking at the signature we get to understand what kind of data can be passed to the function and what we can expect back out of it.

The Question mark following the name parameter specifies that the parameter is optional and we may or may not pass the information for this parameter while calling the function.

**Using the --noImplicitAny**

In case we do not define a type for the parameter, compiler by default assigns the Any type to the parameter. However to avoid such situation and the accidental errors of type mismatches, we can use the –noimplictAny compiler option which we can set in the config file. If we have set this option, even if no type is specified for the parameter, it wont assign the Any type rather it will throw error stating that “Parameter Impilicitly has an any Type”

**Default Initialized Parameters**

In unison with Type Annotations we can assign default values to parameters as well. Lets take the below function

1. function Greetings(salutation : string , name : string=”Priyaranjan”) : string
2. {
3. console.log(salutation+” ”+ name);
4. }

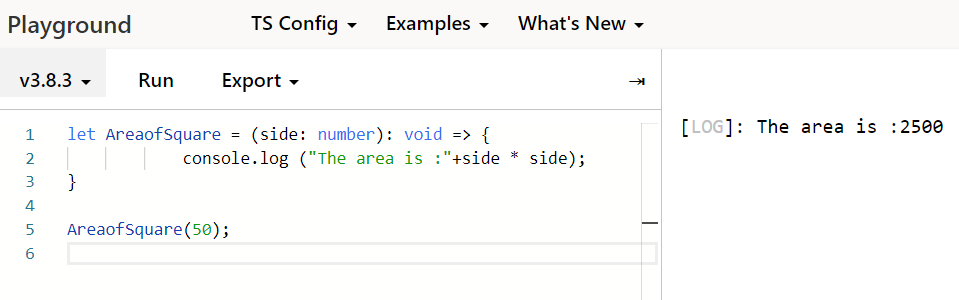
In this function, the name parameter takes in a default value “Priyaranjan” which in effect makes it an optional parameter. In case while calling the function, no value is passed in to the name parameter, it will take the default assigned value. However if some other value is passed to the calling function it will get assigned to the name parameter and will supersede the default value.

# Arrow Functions

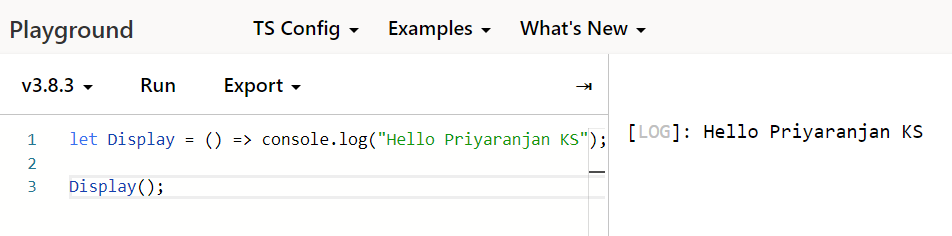
Fat arrow notations are used for anonymous functions. They are also referred to as Lambda functions. The general syntax is as shown below :

1. (param1, param2, ..., paramN) => expression

Using the Arrow notation we can skip the usage of the function keyword. Parameters are passed in the angular brackets <>, and the function expression is enclosed within the curly brackets {}. Lets look at the below example where we have defined the arrow function for calculating the area. Here AreaofSquare is an arrow function. (side:number) denotes the parameter types, :void specifies the return type. The fat arrow => separates the function parameters and the function body. The right side of => can contain one or more code statements.



We can also have parameter less arrow function as shown below :



# Interfaces

Interfaces can be used to create custom types in Typescript. It is a syntactical contract that an object should conform to. Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members.Thus it helps in defining a standard for the deriving classes to follow.

Interfaces are defined with the keyword interface followed by the interface name. Lets look at the below example of interface declaration:

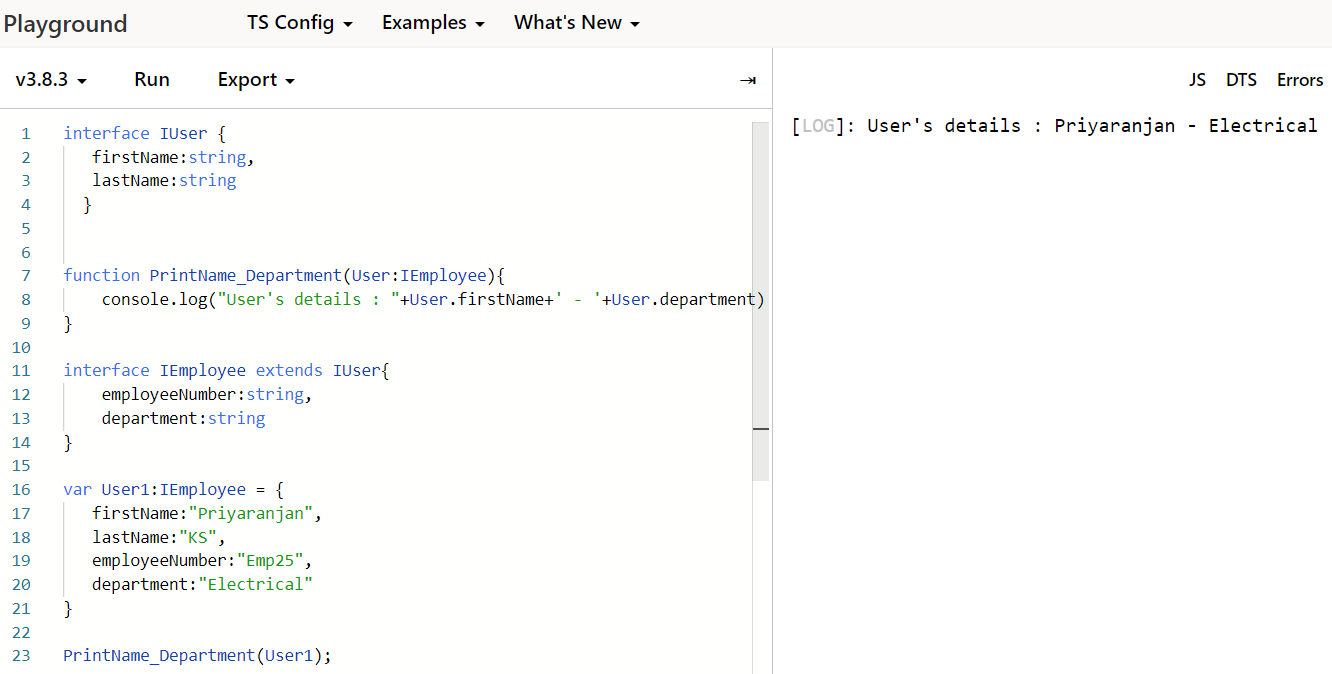
1. interface IUser {
2. firstName:string,
3. lastName:string,
4. }
5. var User1:IUser = {
6. firstName:"Priyaranjan",
7. lastName:"KS",
8. }
9. function PrintName(User:IUser){
10. console.log("User's Full Name : "+User.firstName+' '+User.lastName)
11. }
12. PrintName(User1);



## Inheritance of Interface

An interface can inherit from other interface’s as well. In addition to that, Typescript allows the interface to inherit from multiple interfaces by using the ‘extends’ keyword . A sample of Interface inheritance is shown below :

1. interface IUser {
2. firstName:string,
3. lastName:string
4. }
5. function PrintName\_Department(User:IEmployee){
6. console.log("User's details : "+User.firstName+' - '+User.department)
7. }
9. interface IEmployee extends IUser{
10. employeeNumber:string,
11. department:string
12. }
13. var User1:IEmployee = {
14. firstName:"Priyaranjan",
15. lastName:"KS",
16. employeeNumber:"Emp25",
17. department:"Electrical"
18. }
19. PrintName\_Department(User1);



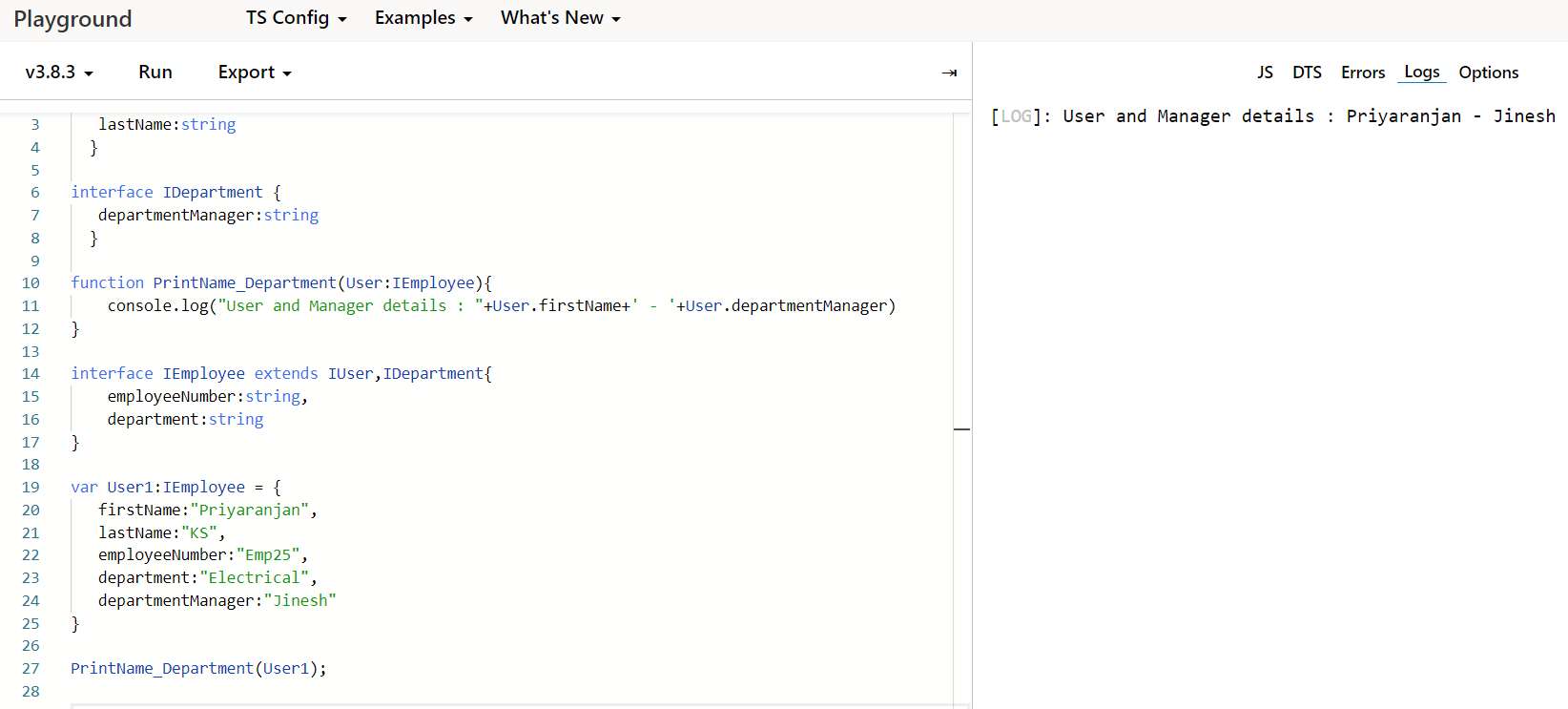
## Multiple Interface Inheritance

If we want to use the members/properties from multiple interfaces it is possible using the below syntax

Interface childInterface extends parentInterface1,parentInterface2{<Signature>}

An example is as shown below :

1. interface IUser {
2. firstName:string,
3. lastName:string
4. }
5. interface IDepartment {
6. departmentManager:string
7. }
8. function PrintName\_Department(User:IEmployee){
9. console.log("User and Manager details : "+User.firstName+' - '+User.departmentManager)
10. }
12. interface IEmployee extends IUser,IDepartment{
13. employeeNumber:string,
14. department:string
15. }
16. var User1:IEmployee = {
17. firstName:"Priyaranjan",
18. lastName:"KS",
19. employeeNumber:"Emp25",
20. department:"Electrical",
21. departmentManager:"Jinesh"
22. }
23. PrintName\_Department(User1);



# Classes

In Object oriented programming a class is defined as a blueprint for an object which serves true for Typescript as well. We can create classes in Typescript using the class keyword. The syntax is as shown below :

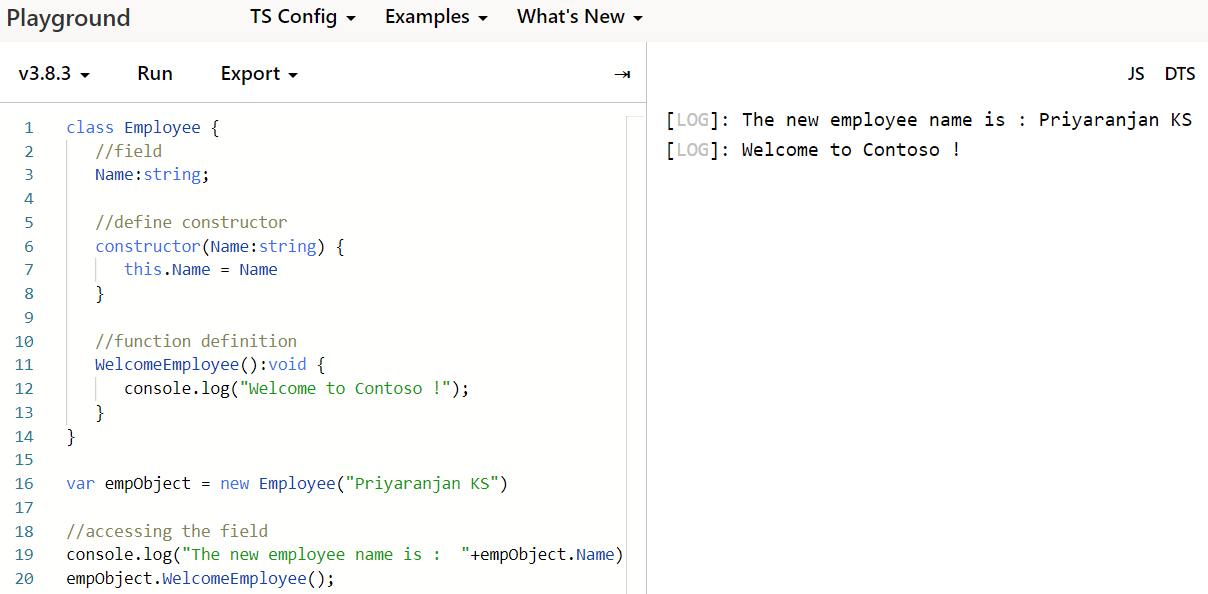
Class className{

//Member and Functions

}

Lets look at the example below :

1. class Employee {
2. //field
3. Name:string;
5. //define constructor
6. constructor(Name:string) {
7. this.Name = Name
8. }
9. //function definition
10. WelcomeEmployee():void {
11. console.log("Welcome to Contoso !");
12. }
13. }
14. var empObject = new Employee("Priyaranjan KS")
15. //accessing the field
16. console.log("The new employee name is :  "+empObject.Name)
17. empObject.WelcomeEmployee();



Here we are decalring a class named employee which has a field named “Name”. We are declaring it without using the var keyord. We are also decalring a constructor for the class. The constructor is a special function of the class which is responsible for initializing the variables of the class. We will be defining the constructor using the “constructor” keyword

ShowEmployeeName is a function that displays the greeting message which is defined without using the function keyword.

To create an instance of the class, use the new keyword followed by the class name. Here we have instantiated the employee class as below. Since we have a parametrized contructor we are passing the parameter as well so that it can be used for initializing class variable “Name”

Var empObject = new Employee(“Priyaranjan KS”)

The class’s attributes and functions can be accessed through the object using the dot(.) notation like

empObject.Name

## Class Inheritance

Just like Interface Inheritance , Classe’s has the ability to create new classes from an existing class. The class that is extended to create newer classes is called the parent class/super class and the newly created classes are called the child/sub classes.

We can use the Extends key word to define the inheritance as shown below. Here the Greetings class inherits from the Employee class which has the field name declared. When an object of Greetings is instantiated, it can still access the Employee class attributes and functions.



# Getting Started with SharePoint Framework Development

SharePoint Framework is the new development model in which lots of work had been going on in the past year. It went to General Availability on Feb 23rd 2017. It is a page and web part model that provides full support for client-side SharePoint development, easy integration with SharePoint data and support for open source tooling. With the SharePoint Framework, you can use modern web technologies and tools in your preferred development environment to build productive experiences and apps in SharePoint.

# Set up SharePoint Framework Development Environment

Let us see how to set up the development environment so that we can kick start with SharePoint Framework development. Below are the required components that we will have to install in the environment.

* Node JS
* Yeoman and Gulp
* Yeoman SharePoint Generator
* Code Editor(Visual Studio Code/Webstorm)
* Postman and Fiddler(optional)

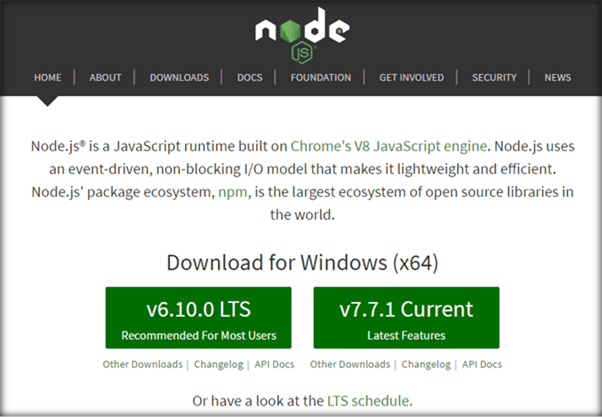
### Install Node JS



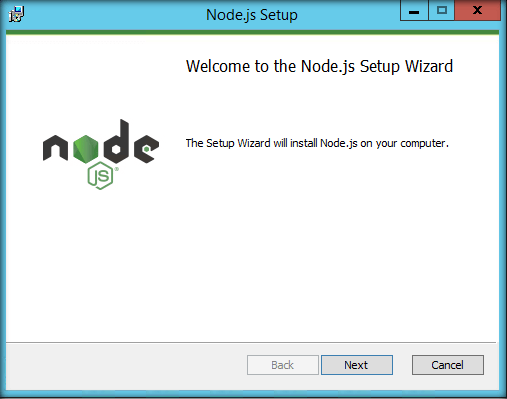
Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It uses an event-driven, non- blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the

largest ecosystem of open source libraries in the world. We will be making use of npm along with Yeoman and Gulp to package and deploy modules.

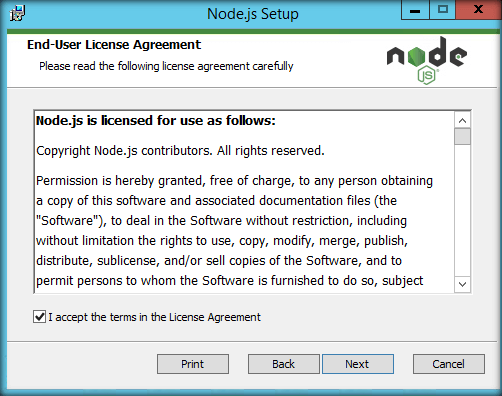
As the first step we will install NodeJS Long Term Support Version (LTS). We can install Node JS from this [link](https://nodejs.org/en/) .



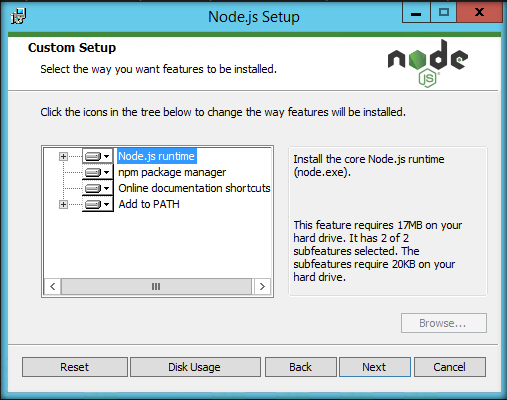
Once we have downloaded the LTS version, run the executable file and proceed.



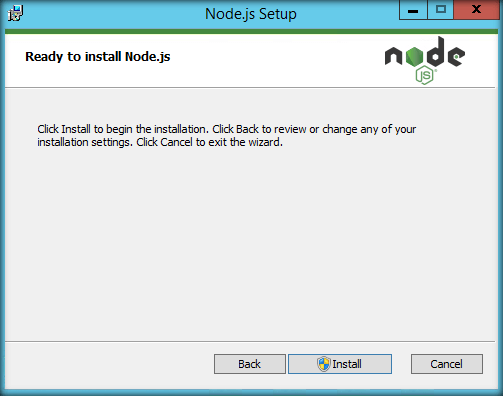
Accept the license agreement and click on Next.



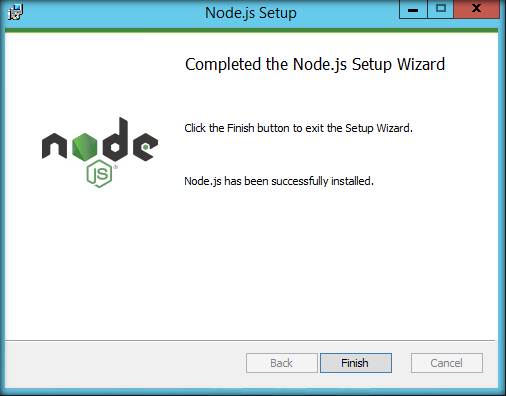
We will select Node.js run time installation.



Click on Install to start the installation procedure.

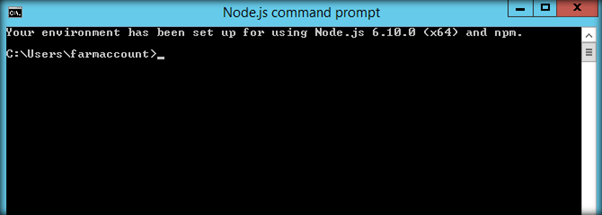


Finally, we are done installing NodeJS.

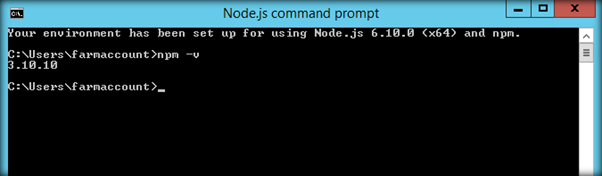


Click on Finish button and restart your computer. You won’t be able to run Node.js until you restart your computer.

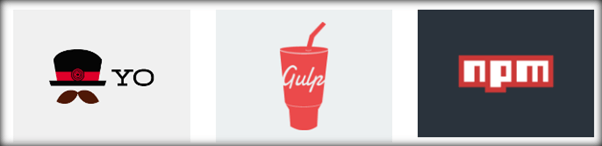
If we run the NodeJS command prompt, we will get the message as shown below. Thus, the Node JS has been successfully installed in the local machine.



Now, let’s see the version of Node Package Manager (npm) by running the command **npm –v** . It is running V3 version.



### Install Yeoman and Gulp



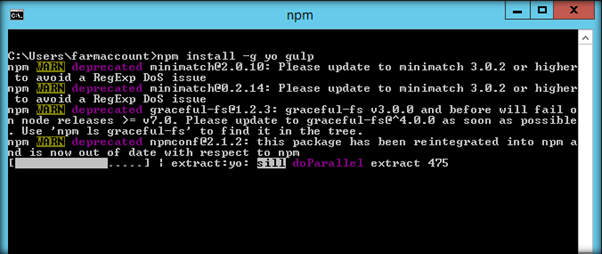
[Yeoman](http://yeoman.io/) is a scaffolding tool for modern web apps. It helps you to kick-start new projects, prescribing best practices and tools to help you stay productive. Often called Yo, it scaffolds out a new

application, writing your build configuration (e.g Gulpfile) and pulling in relevant build tasks and package manager dependencies (e.g npm) that you might need for your build.

Gulp is a JavaScript task runner that helps us automate common tasks like refreshing your browser when you save a file, Bundling and minifying libraries and CSS, Copying modified files to an output directory etc. We will be using Yo and Gulp together for creating SharePoint Client Webparts.

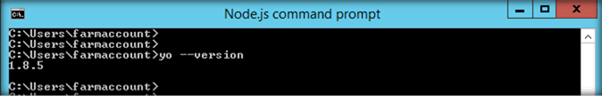
Now, let’s install Yeoman and Gulp simultaneously by running the below command:

*npm install -g yo gulp*



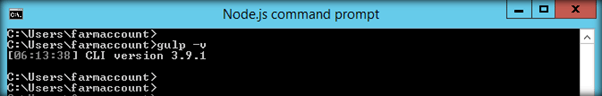
We can get the version of Yeoman using the command:

*yo --version*



We can get the Gulp Version using the command:

*gulp –v*

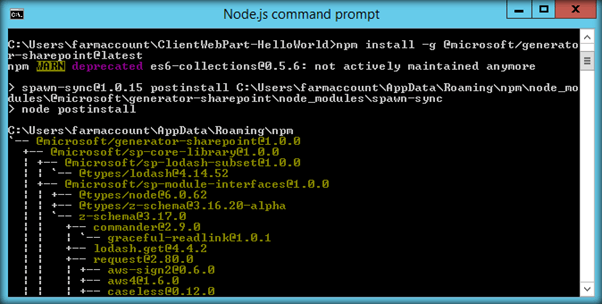


### Install Yeoman SharePoint Generator

The Yeoman SharePoint web part generator helps you to quickly create a SharePoint client-side solution project with the right tool chain and project structure.

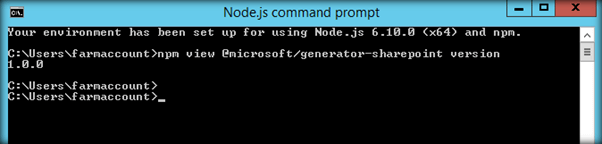
Yeoman SharePoint Generator can be installed using the below command:

*npm install -g @microsoft/generator-sharepoint@latest*



We can get the version of Yeoman Generator by running the below command. As we can see 1.0.0 indicates General Availability version.

*npm view @microsoft/generator-sharepoint version*

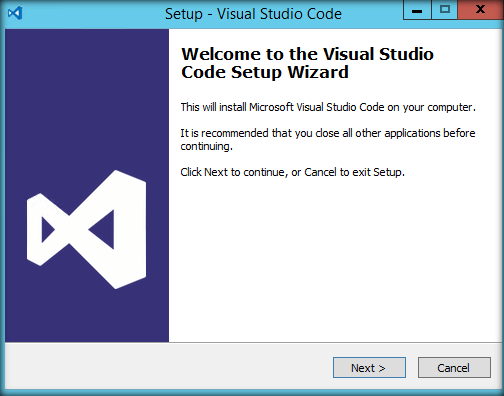


### Code Editor

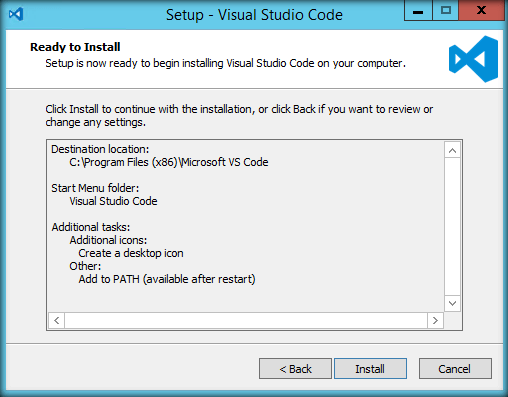
Next, we need a code editor that will help us with code editing. We can use any code editor or IDE that supports client-side development to build our web part, such as:

* Visual Studio Code
* Atom
* Webstorm

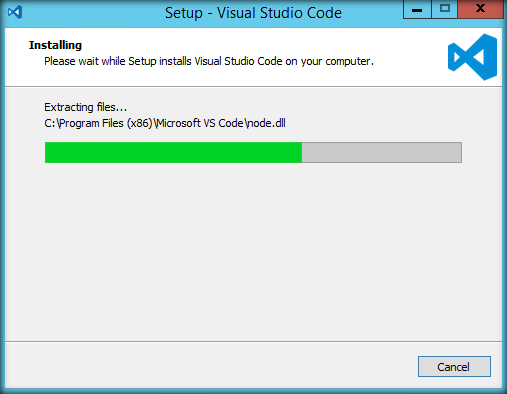
We will use Visual Studio Code in this walkthrough. You can get it from [here](https://code.visualstudio.com/download) .



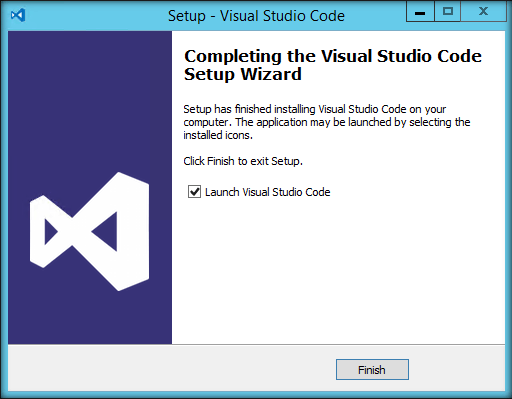
Once we have downloaded the exe, proceed with the installation.



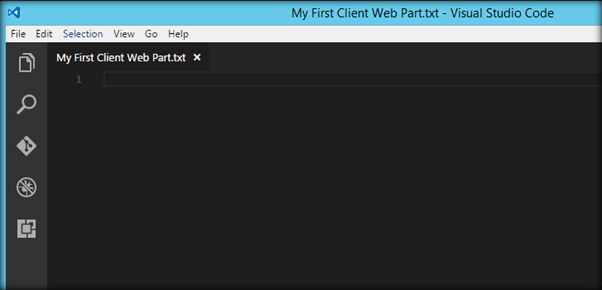
Click on Install to start the installation procedure.



Finally, we have completed installation of the Visual Studio Code editor.



Sample Screen Shot,



### Additional Tools for Development and Debugging

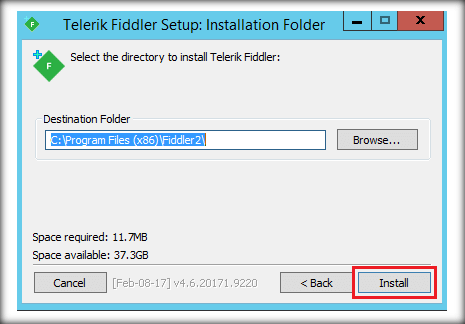
Once we start the development, we must debug or test the application. Fiddler and Postman can help us in this task.

Fiddler

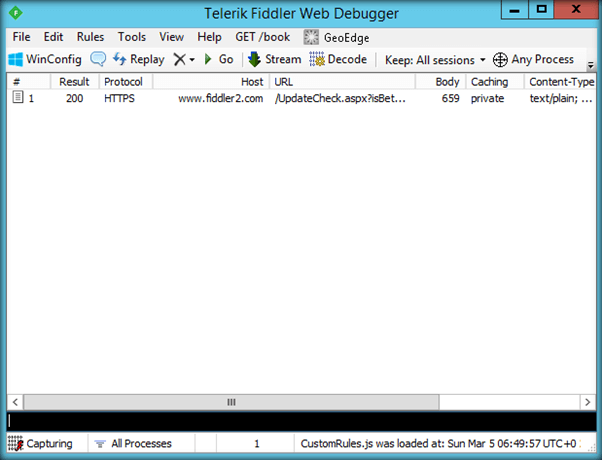
Fiddler is an HTTP debugging proxy server application. It captures HTTP and HTTPS traffic and logs it for the user to review. You can get fiddler from [here](https://www.telerik.com/download/fiddler)



Once the executable has been downloaded. Click on Install to set up Fiddler in your local machine.

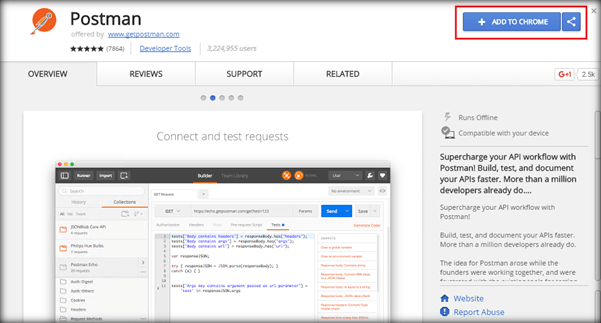


Using fiddler, we can examine the traffic as it is being sent or received.

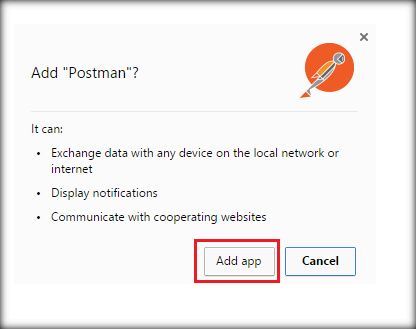


## Postman

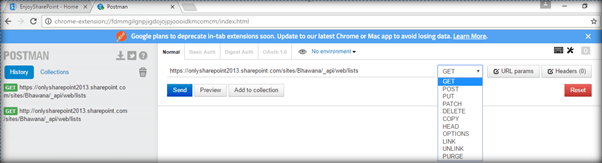
Postman can be used to test SharePoint’s REST service endpoints and verify the returned data and request headers. We can get Postman from [here](https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en)



Postman can be added to Chrome as an app.



The REST URL can be entered in the Request URL field and we can click on Send to get the SharePoint data.



Thus, we saw how to set up the environment and now we are ready to get started with the new SharePoint Framework development model.

# Getting Started with SharePoint Framework Development using TypeScript

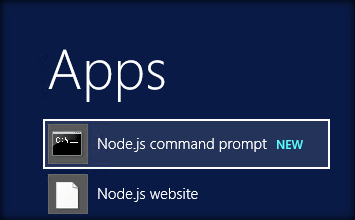
In the first section of the book we will be using TypeScript to build SharePoint Framework client webparts. As we proceed further, we will see how to use React JS and PnP JS to build the SPFx webparts

# Create the First Hello World Client Web part

In this section, we will see how to create and deploy the first client web part using SharePoint Framework. We will be creating a Hello World client web part using TypeScript to understand the project structure and testing procedure.

### Create the Web part project

Before moving forward, ensure that the SharePoint Framework development environment is ready. Spin up Node.js command prompt using which we will be creating the web part project structure.

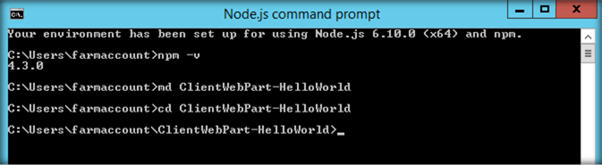


We can create the directory where we would be adding the solution using the below command:

*md ClientWebPart-HelloWorld*

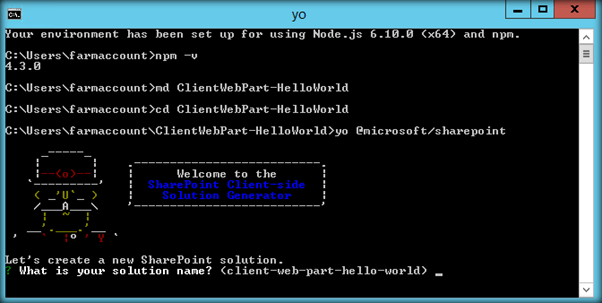
Let’s move to the newly created working directory using the command:

*cd ClientWebPart-HelloWorld*



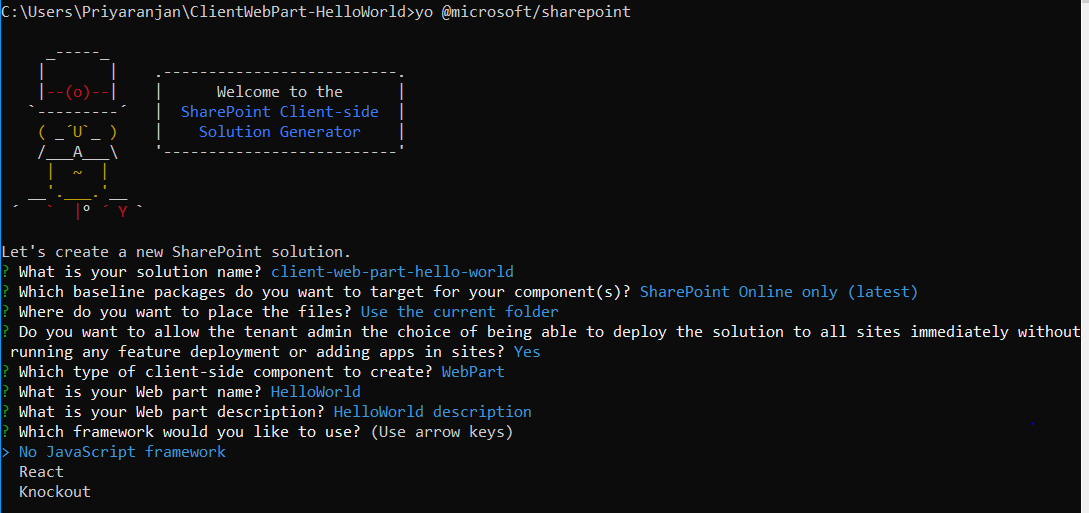
We will then create the client web part by running the Yeoman SharePoint Generator:

*yo @microsoft/sharepoint*



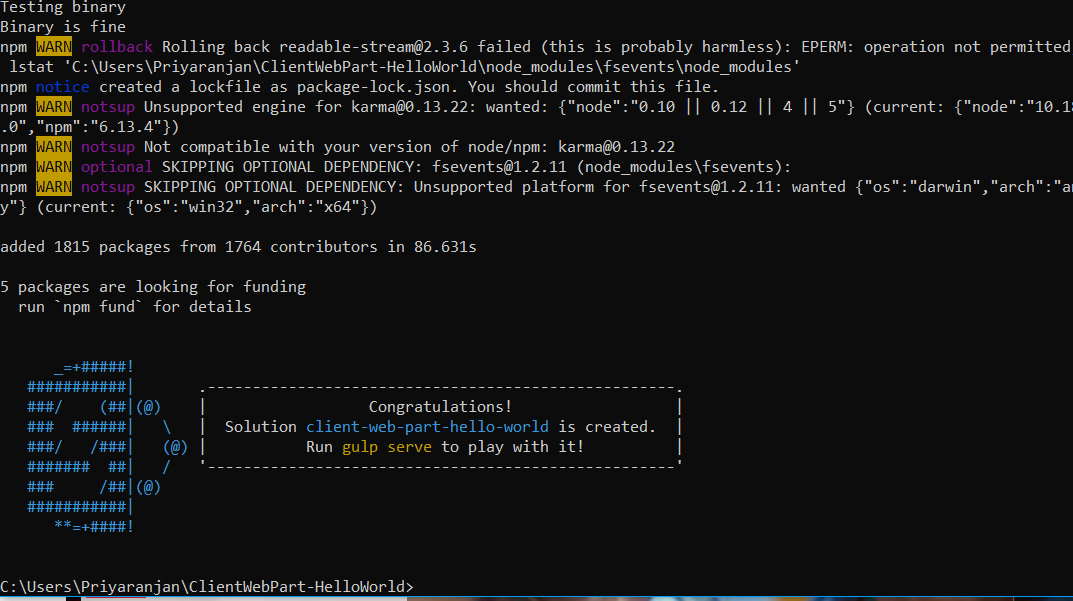
This will display the prompt which we will have to fill up so as to proceed with project creation,

* What is your solution name? : Accept the default client-web-part-hello-world as your solution name and choose Enter.
* Where do you want to place your files : Use Current Folder
* What framework would you like to start with : Select “No javaScript web framework” for the time being as this is a sample web part



* What is your webpart name: Go on and press enter to accept the default Web part name as HelloWorld
* Go on and press enter to accept the default Web part description as HelloWorld description

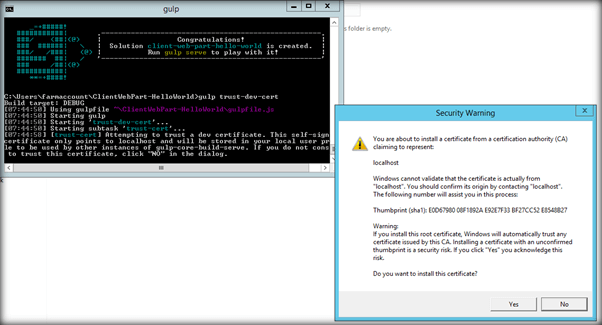
Yeoman has started working on the creation of the project. It will install the required dependencies and scaffold the solution files for the HelloWorld web part which will take some time to complete. Once completed, we will get a Congratulations message.

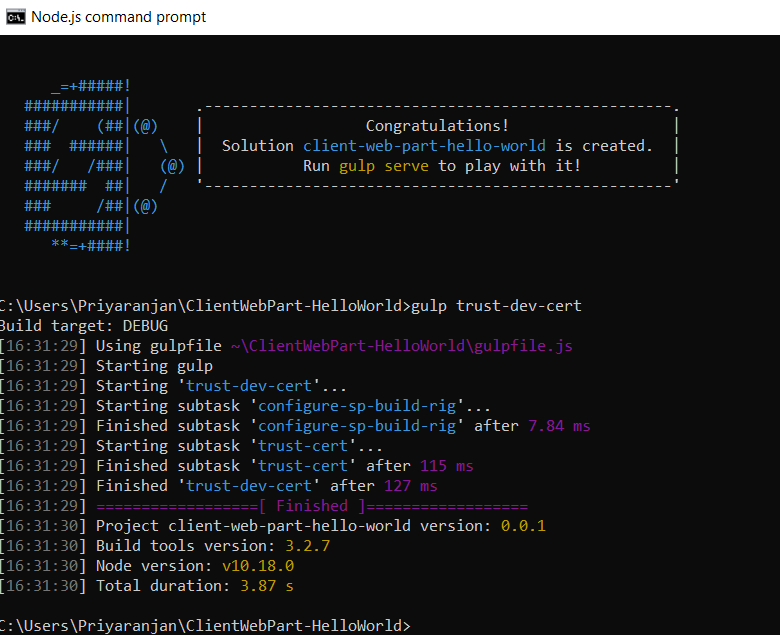


### Test the web part

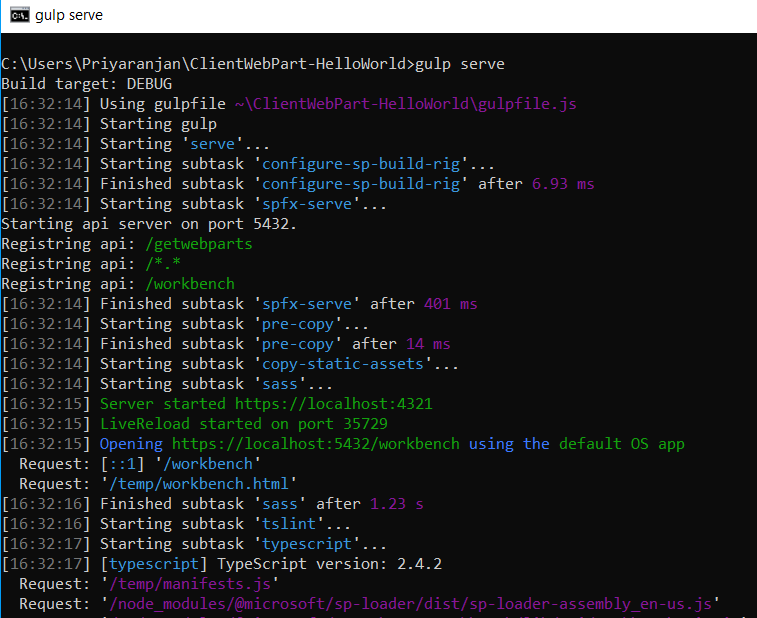
To test the client web part, we can build and run it on the local web server where we are developing the web part. SharePoint Framework development uses HTTPS endpoint by default. Since a default certificate is not configured for the local development environment, our browser will report a certificate error. The SharePoint Framework tool chain comes with a developer certificate that we can install for testing client web parts locally. From the current web part directory, run the below command:

*gulp trust-dev-cert*



Click on Yes to install the certificate.

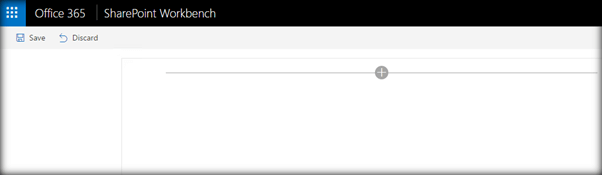
Now, let’s preview the web part by running the *gulp serve* command. This command will execute a series of gulp tasks and will create a Node-based HTTPS server at 'localhost:4321'. It will then open the browser and display the client web part.



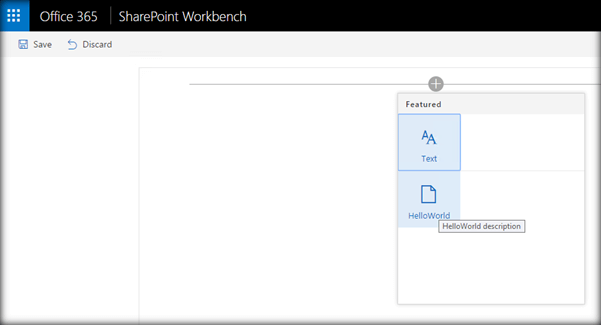
### SharePoint Workbench

SharePoint Workbench is a developer design surface that enables us to test the developed client web parts without deploying them directly to SharePoint. It provides a client-side page to which we can add the created web parts.

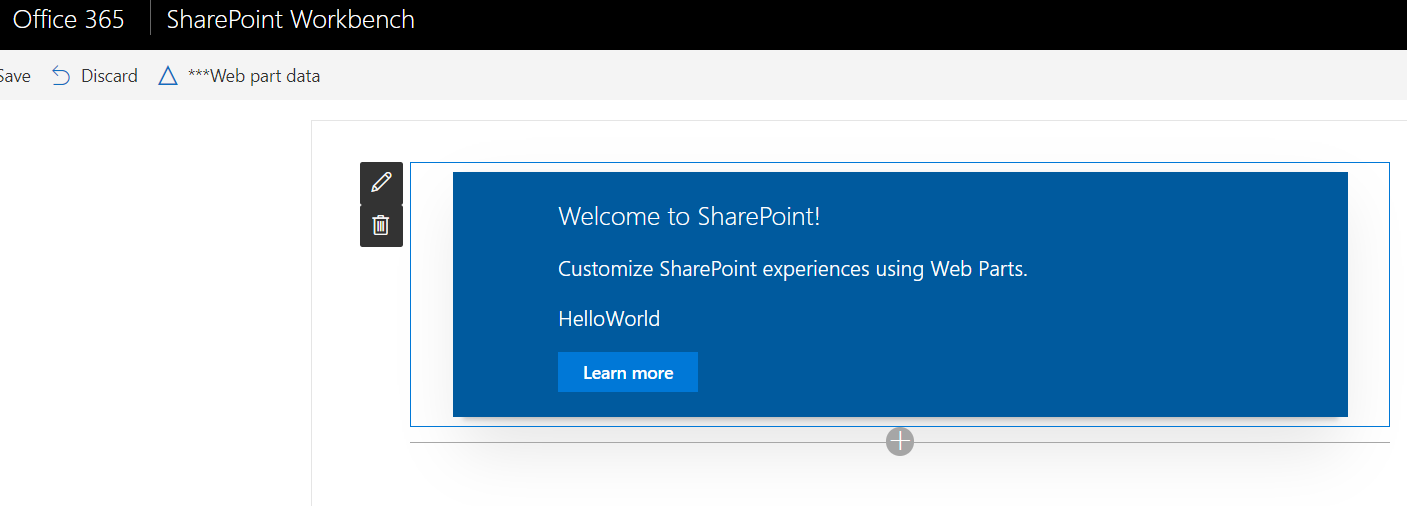
Thus, the SharePoint Workbench has opened in the browser but there are no visible web parts. So, let’s go ahead and click on the Plus sign.



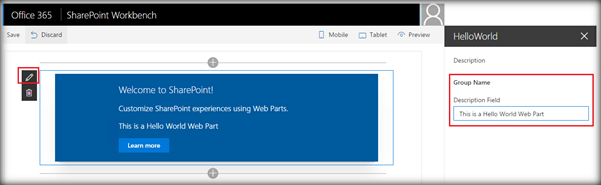
It will give us the option to add the Hello World web part that we have created recently.



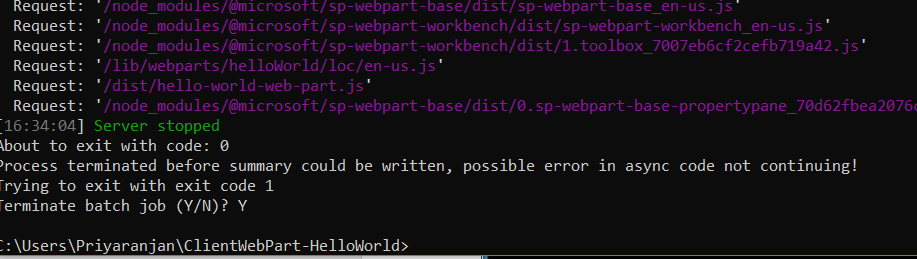
On clicking it, the web part will be added to the page. The web part contains few custom messages.



We can edit the description property directly from the UI as shown below. However, if we want to edit this web part to add more details and functionality, we must go back and terminate the gulp server command.

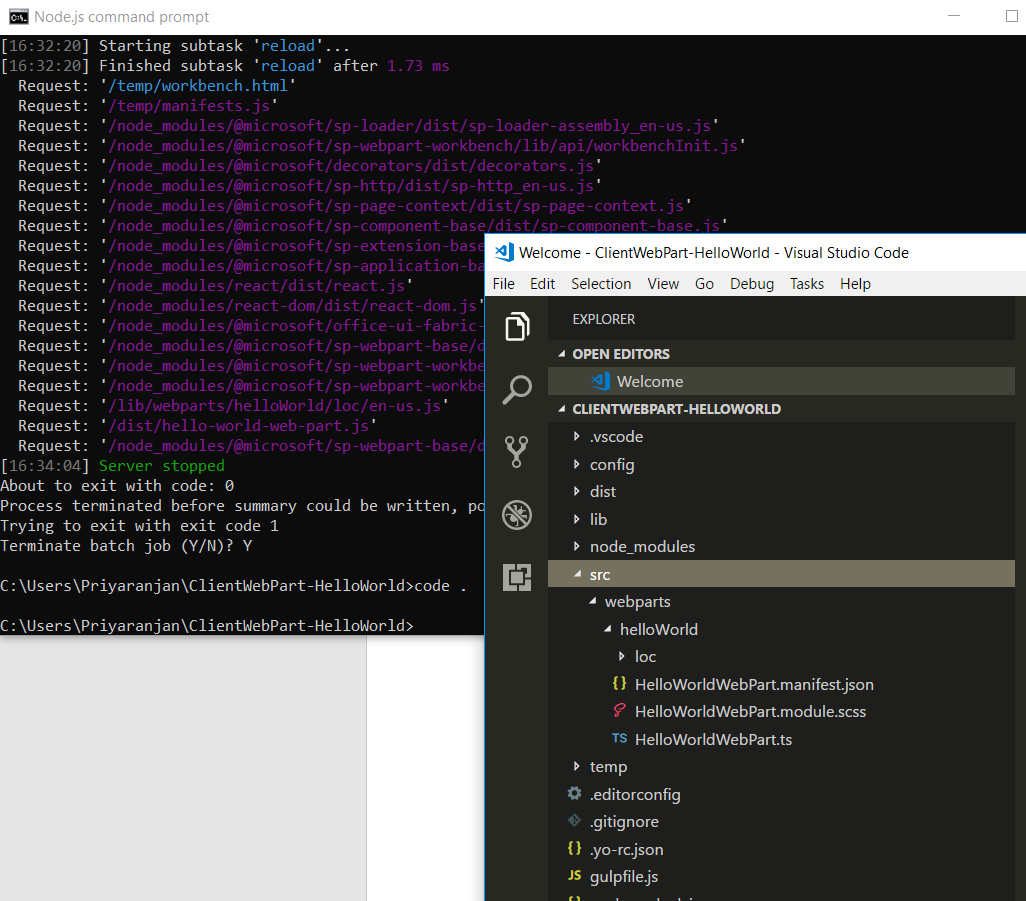


To stop Gulp from listening to the process we can press ‘Control + C’. This will terminate the Gulp Serve command and stop the server.

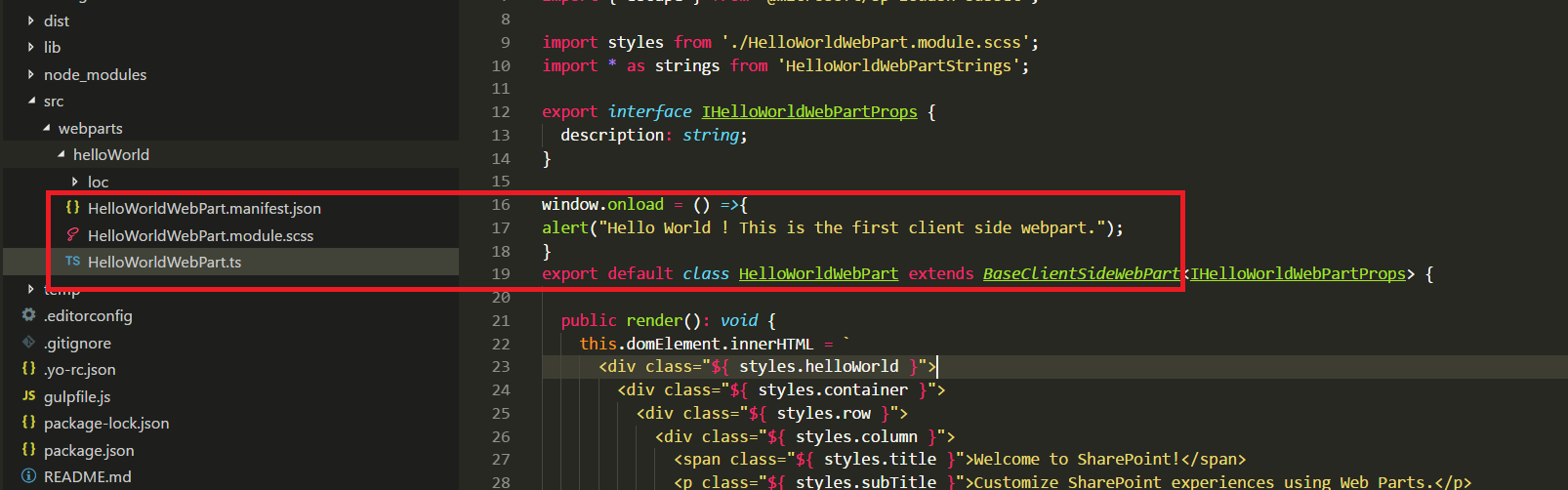


### Edit the web part

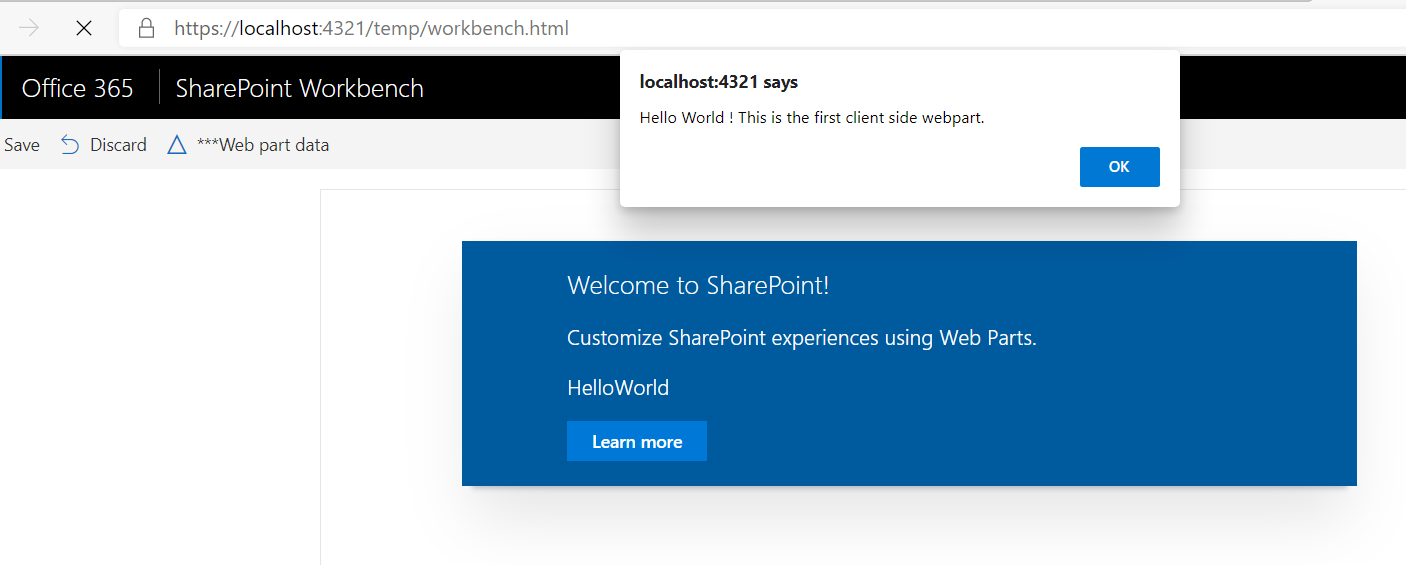
Now, let’s try to edit the web part and add more functionality to it. To do that, navigate to ‘src\webparts\helloWorld’ location.Run ‘Code .’ in the console which will open up the Visual Studio Code editor window.



In the left pane of Visual Studio Code, we can see the project structure. The bulk of the logic resides within the *HelloWorldWebPart.ts* file. Let’s add JavaScript code to alert a message within this typescript file.

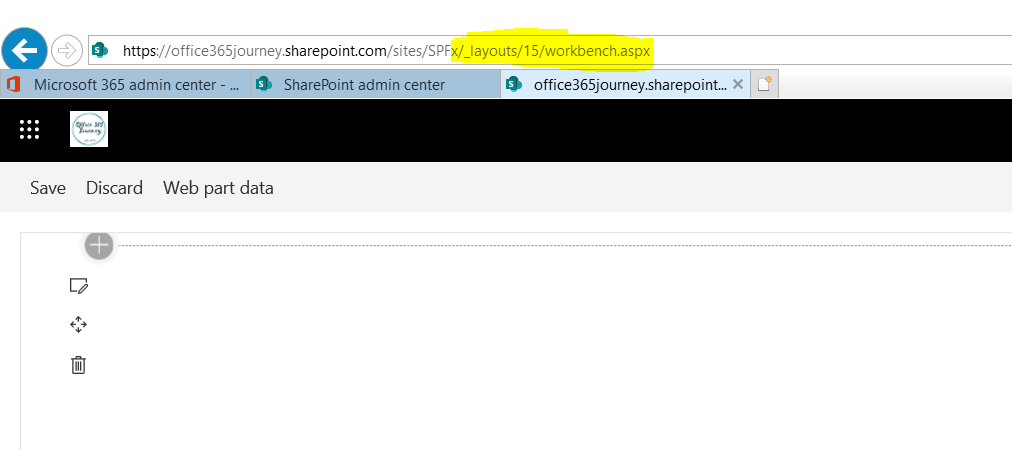


On clicking save Gulp will rebuild the web part project . Again, running ‘gulp serve’ will display the updated web part in the browser. This time it will display the alert message as well.

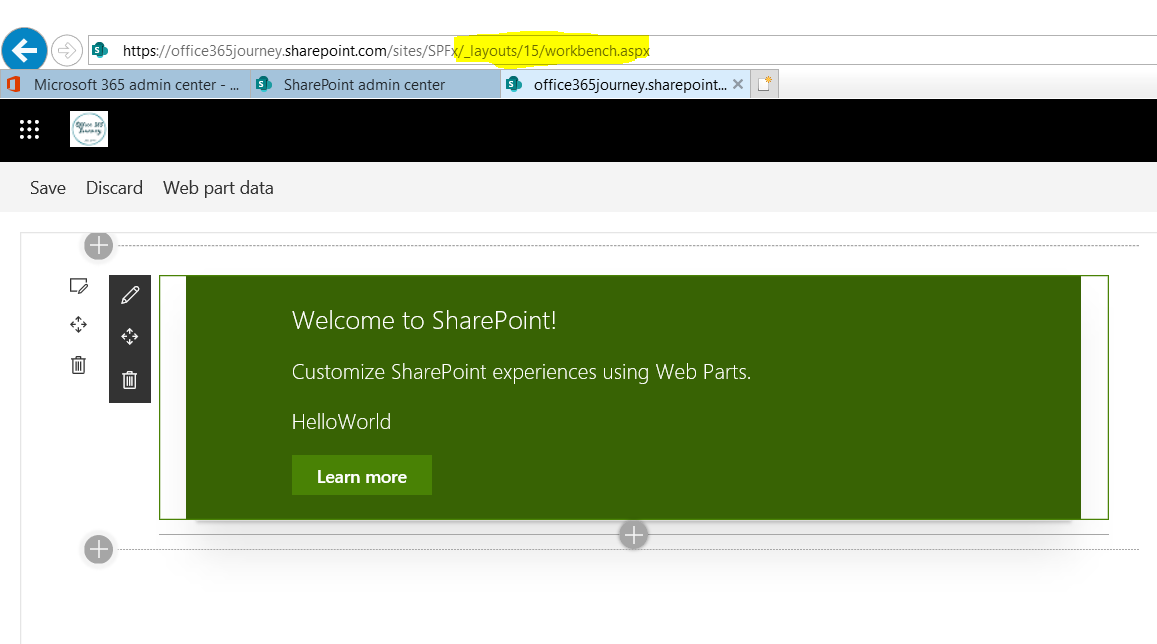


### Add the web part to SharePoint

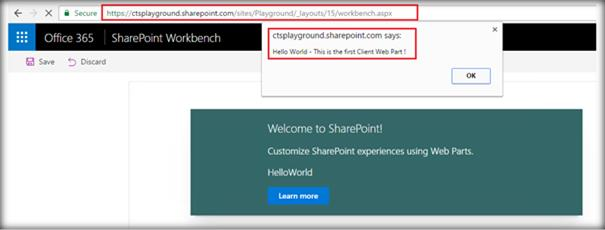
So far we were testing the web part in SharePoint Workbench locally, now let’s try to test it within the SharePoint Context. SharePoint Workbench is also hosted in SharePoint Online to preview the web part. It can be accessed by adding ‘ \_layouts/15/workbench.aspx’ to the SharePoint Online URL.



Expand the Plus sign and add the Hello World web part.



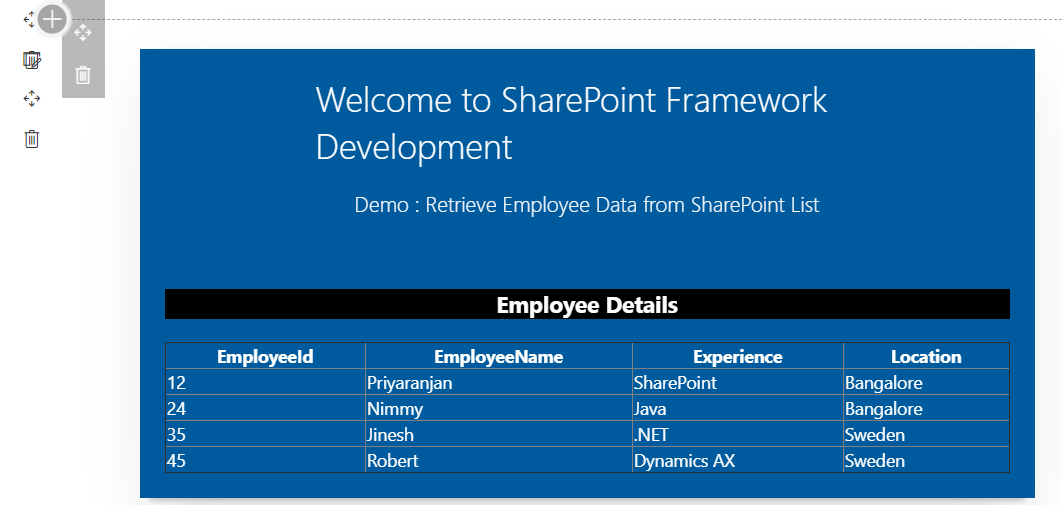
The web part has triggered the alert message in the page indicating successful hosting of the web part within SharePoint.



Thus, we saw how to create a client web part using SharePoint Framework and test it within SharePoint Online.

# Create SharePoint Framework Client Web Part to Retrieve and Display List Items

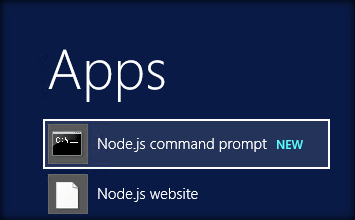
In this section, we will be creating a client Web part using TypeScript, which will be retrieving the list items from SharePoint List (EmployeeList) and display it in the tabular form in the client Web part, as shown below.



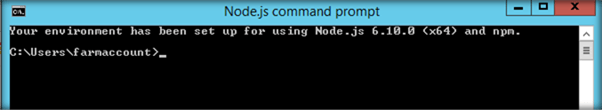
The Solutions files used in this section is zipped and uploaded to the Microsoft [TechNet gallery](https://gallery.technet.microsoft.com/Retrieve-SharePoint-List-24805125). Feel free to download it.

### Create the Web part Project

Spin up Node.js command prompt, using which we will be creating the Web part project structure.



This will open the console where we can create the SharePoint Framework project structure.

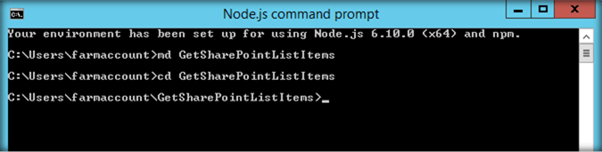


We can create the directory, where we will be adding the solution, using the command given below.

*md GetSharePointListItems*

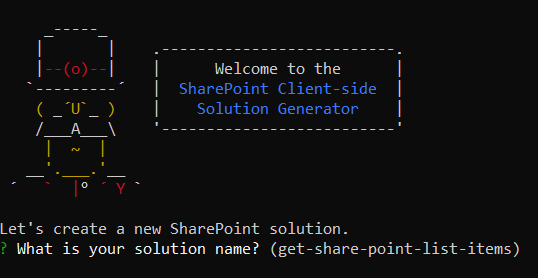
Let’s move to the newly created working directory, using the command.

*cd GetSharePointListItems*



We will then create the client Web part by running the Yeoman SharePoint Generator.

*yo @microsoft/sharepoint*



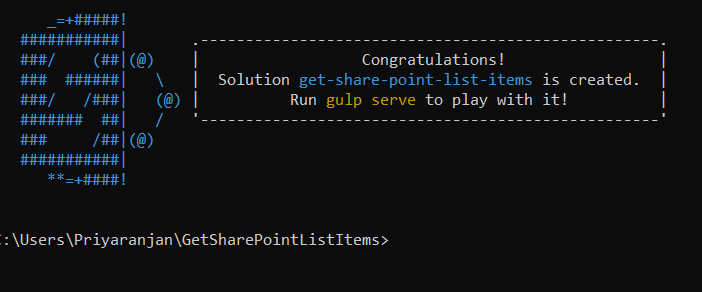
This will display the prompt, which we must fill up, to proceed with the project creation.

* What is your solution name? : Set it to ‘GetSPListItems’.

On pressing enter, we will be asked to chose the working folder for the project.

* Where do you want to place your files- Use current folder.
* What framework would you like to start with- Select “No javaScript web framework” for the time being, as this is a sample Web part.
* What is your Webpart name- We will specify it as ‘GetSPListItems’ and press Enter
* What is your Webpart description- We will specify it as this Webpart will retrieve the list items from SharePoint list and display in a table

Yeoman has started working on the scaffolding of the project. It will install the required dependencies and scaffold the solution files for the ‘GetListItems’ Web part, which will take some time to complete. Once completed, we will get a congratulations message.



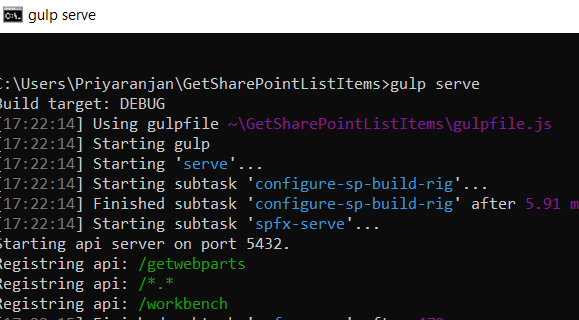
### Test the Web part locally

To test the client Web part, we can build and run it on the local Web Server, where we are developing the Web part. SharePoint Framework development uses HTTPS endpoint by default. Since a default certificate is not configured for the local development environment, our Browser will report a

certificate error. SharePoint Framework tool chain comes with a developer certificate, which we can install for testing the client Web parts locally. From the current Web part directory, run the command given below.

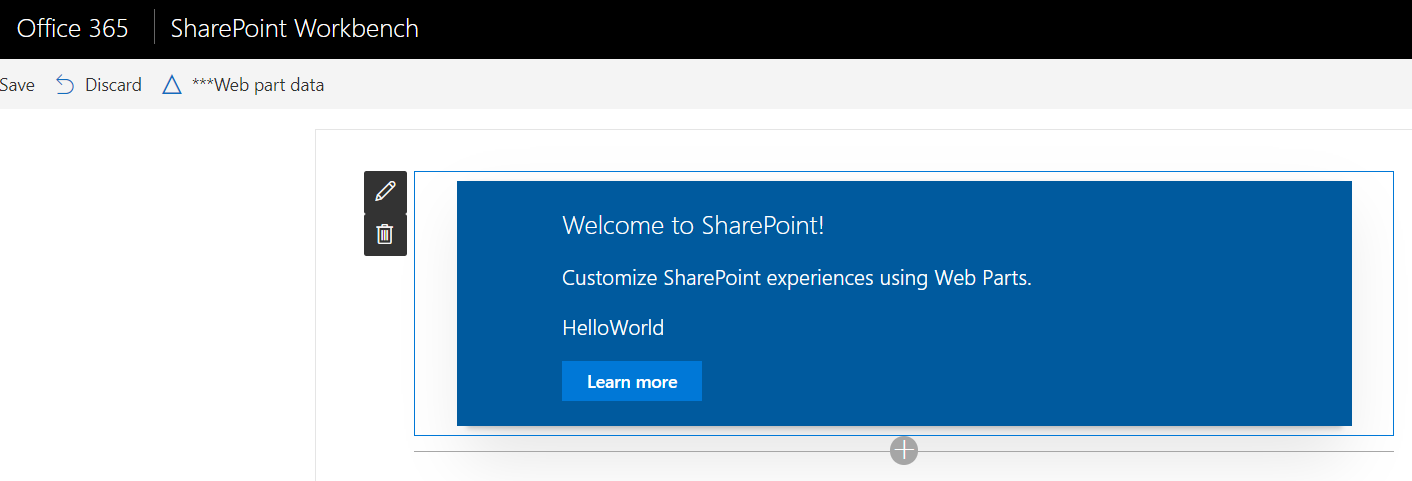
*gulp trust-dev-cert*

Now, let’s preview the Web part by running the *gulp serve* command.



This command will execute a series of gulp tasks and will create a Node-based HTTPS Server at 'localhost:4321'. It will then open the Browser and display the client Web part.

This indicates that the project structure is set up correctly. We will now open the solution in Visual Studio Code to add the logic to retrieve the list items from SharePoint and display it as a table in this page.

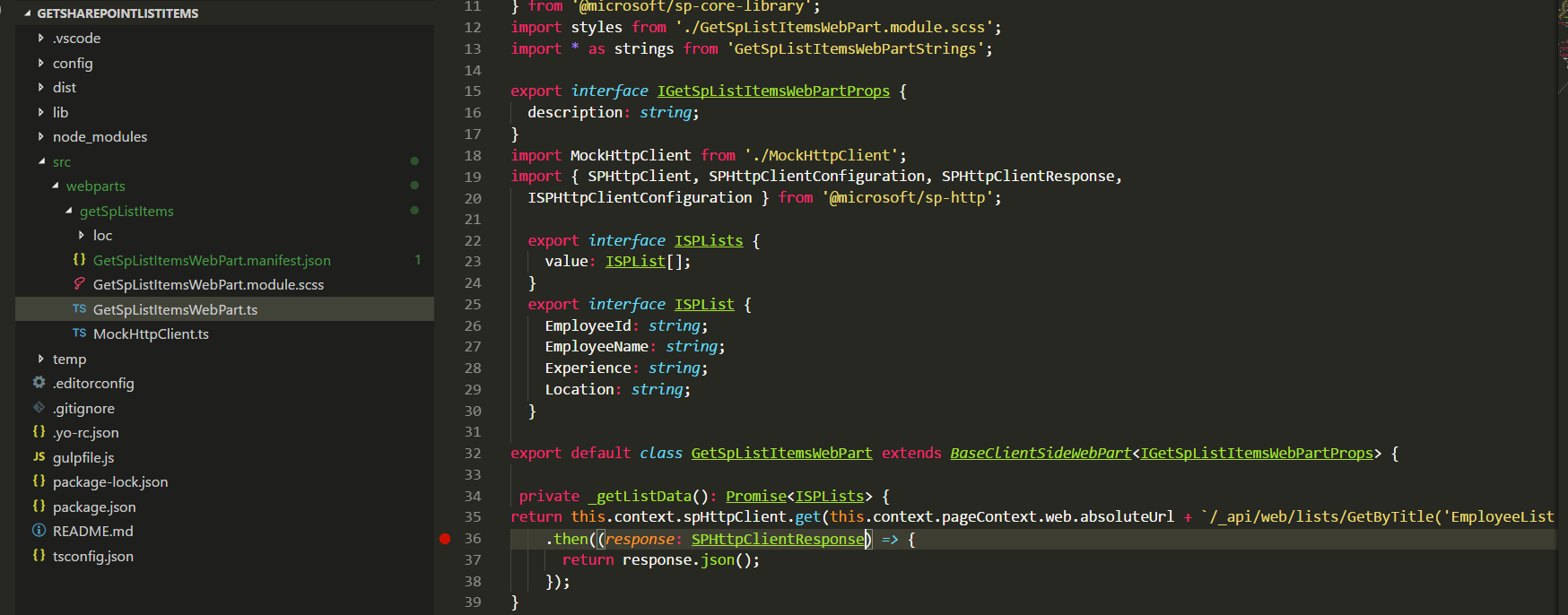


To stop Gulp from listening to the process, we can press ‘*Control + C’*. This will terminate the Gulp Serve command and stop the Server.

### Edit the web part

Now let’s try to edit the web part and add more functionality to it. To do that navigate to “src\webparts\getSpListItems” location.

In the left pane of Visual Studio Code, we can see the project structure. The bulk of the logic resides within the *GetSPListItemsWebPart.ts* file. Let’s add the code to retrieve SharePoint list items from the Employee List within this TypeScript file.



### Define List Model

Since we want to retrieve an Employee list items data, we will be creating list model with SharePoint list fields in the GetSpListItemsWebPart.TS file, as shown below. Place it above the ‘GetSpListItemsWebPart’ class.

1. export interface ISPLists {
2. value: ISPList[];

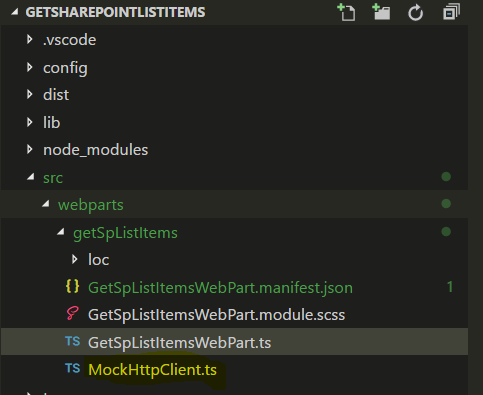
3. }

1. export interface ISPList {
2. EmployeeId: string;
3. EmployeeName: string;
4. Experience: string;
5. Location: string;

9. }

### Create Mock HTTPClient to test data locally

To test the list item retrieval in the local workbench, we will create a mock store, which returns mock Employee list data. We will create a new file inside ‘src\webparts\ getSpListItems’ folder named MockHttpClient.ts, as shown below.



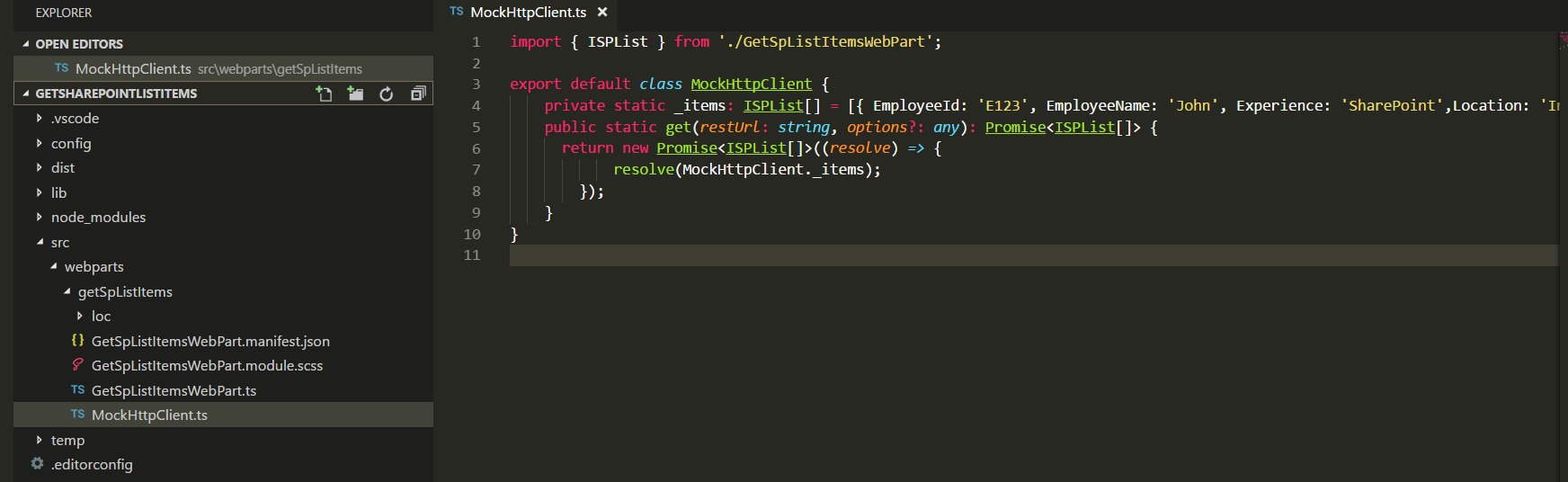
We will then copy the code given below into MockHttpClient.ts, as shown below.

1. import { ISPList } from './GetSpListItemsWebPart'; 2.

1. export default class MockHttpClient {
2. private static \_items: ISPList[] = [{ EmployeeId: 'E123', EmployeeName: 'John', Experience: 'SharePoint',Location:'India' },];
3. public static get(restUrl: string, options?: any): Promise<ISPList[]> {
4. return new Promise<ISPList[]>((resolve) => {
5. resolve(MockHttpClient.\_items); 8. });

9. }

10. }



We can now use the MockHttpClient class in the ‘GetSPListItems’ class. Let’s import the ‘MockHttpClient’ module by going to the GetSpLitItemsWebPart.ts and pasting the line given below just after “import { IGetSpListItemsWebPartProps } from './IGetSpListItemsWebPartProps';”

1. import MockHttpClient from './MockHttpClient';

We will also add the mock list item retrieval method within the ‘GetSpListItemsWebPart’ class.

1. private \_getMockListData(): Promise<ISPLists> {
2. return MockHttpClient.get(this.context.pageContext.web.absoluteUrl).then(() => {
3. const listData: ISPLists = {
4. value:

5. [

1. { EmployeeId: 'E123', EmployeeName: 'John', Experience: 'SharePoint',Location: 'India' },
2. { EmployeeId: 'E567', EmployeeName: 'Martin', Experience: '.NET',Location: 'Qatar' },
3. { EmployeeId: 'E367', EmployeeName: 'Luke', Experience: 'JAVA',Location: 'UK' }

9. ]

10. };

1. return listData;
2. }) as Promise<ISPLists>;

13. }

### Retrieve SharePoint List Items

SharePoint Framework has the helper class spHttpClient, which can be utilized to call REST API requests against SharePoint. We will use REST API: “/\_api/web/lists/GetByTitle('EmployeeList')/Items” to get the list items from SharePoint List.

To use ‘spHttpClient’, we will first have to import it from the ‘@microsoft/sp-http’ module. We will import this module by placing the line given below after the mockHttpClient import code -*“import MockHttpClient from './MockHttpClient';*”

import { SPHttpClient, SPHttpClientConfiguration, SPHttpClientResponse,

ISPHttpClientConfiguration } from '@microsoft/sp-http';

We will be then adding the method given below to get SharePoint list items, using REST API within the ‘GetSpListItemsWebPart’ class.

1. private \_getListData(): Promise<ISPLists> {
2. return this.context.spHttpClient.get(this.context.pageContext.web.absoluteUrl +

`/\_api/web/lists/GetByTitle('EmployeeList')/Items`, SPHttpClient.configurations.v1)

1. .then((response: SPHttpClientResponse) => {
2. debugger;
3. return response.json();

6. });

7. }

### Render the SharePoint List Items from Employee List

Once we run the gulp serve command, we can test the Web part in SharePoint Workbench in the local environment or using SharePoint Online Context. SharePoint Framework uses ‘EnvironmentType’ module to identify the environment, where the Web part is executed.

In order to implement this, we will import ‘Environment’ and the ‘EnvironmentType’ modules from the @microsoft/sp-core-library bundle by placing it at the top of the GetSpListItemsWebpart.ts file.

import { Environment, EnvironmentType

} from '@microsoft/sp-core-library';

We will then check Environment.type value and if it is equal to Environment.Local, the MockHttpClient method, which returns dummy data which will be called else the method that calls REST API which can retrieve SharePoint list items will be called.

1. private \_renderListAsync(): void { 2.

1. if (Environment.type === EnvironmentType.Local) {
2. this.\_getMockListData().then((response) => {
3. this.\_renderList(response.value);

6. });

7. }

1. else {
2. this.\_getListData()
3. .then((response) => {
4. this.\_renderList(response.value); 12. });

13. }

14. }

Finally, we will add the method given below, which will create HTML table out of the retrieved SharePoint list items.

1. private \_renderList(items: ISPList[]): void {
2. let html: string = '<table class="TFtable" border=1 width=100% style="border- collapse: collapse;">';
3. html +=

`<th>EmployeeId</th><th>EmployeeName</th><th>Experience</th><th>Location</th>`;

1. items.forEach((item: ISPList) => {
2. html += `
3. <tr>
4. <td>${item.EmployeeId}</td>
5. <td>${item.EmployeeName}</td>
6. <td>${item.Experience}</td>
7. <td>${item.Location}</td>

11. </tr>

12. `;

13. });

1. html += `</table>`;
2. const listContainer: Element = this.domElement.querySelector('#spListContainer');
3. listContainer.innerHTML = html;

17. }

To enable rendering of the list items given above, we will replace Render method in the ‘GetSpListItemsWebPart’ class with the code given below.

1. public render(): void {
2. this.domElement.innerHTML = `
3. <div class="${styles.helloWorld}">
4. <div class="${styles.container}">
5. <div class="ms-Grid-row ms-bgColor-themeDark ms-fontColor-white ${styles.row}">
6. <div class="ms-Grid-col ms-u-lg10 ms-u-xl8 ms-u-xlPush2 ms-u-lgPush1">
7. <span class="ms-font-xl ms-fontColor-white" style="font-size:28px">Welcome to SharePoint Framework Development</span>

8.

1. <p class="ms-font-l ms-fontColor-white" style="text-align: center">Demo : Retrieve Employee Data from SharePoint List</p>
2. </div>
3. </div>
4. <div class="ms-Grid-row ms-bgColor-themeDark ms-fontColor-white ${styles.row}">
5. <div style="background-color:Black;color:white;text-align: center;font-weight: bold;font-size:18px;">Employee Details</div>
6. <br>
7. <div id="spListContainer" />
8. </div>
9. </div> 18. </div>`;

19. this.\_renderListAsync();

20. }

### TS File Contents

The code contents used in the TS file to retrieve and display list items are given below:

1. import { Version } from '@microsoft/sp-core-library';
2. import {
3. BaseClientSideWebPart,
4. IPropertyPaneConfiguration,
5. PropertyPaneTextField
6. } from '@microsoft/sp-webpart-base';
7. import { escape } from '@microsoft/sp-lodash-subset'; 8.
8. import {
9. Environment,
10. EnvironmentType
11. } from '@microsoft/sp-core-library'; 13.

14.

1. import styles from './GetSpListItems.module.scss';
2. import \* as strings from 'getSpListItemsStrings';
3. import { IGetSpListItemsWebPartProps } from './IGetSpListItemsWebPartProps';
4. import MockHttpClient from './MockHttpClient'; 19.
5. import {
6. SPHttpClient
7. } from '@microsoft/sp-http'; 23.

24.

1. export interface ISPLists {
2. value: ISPList[];

27. }

1. export interface ISPList {
2. EmployeeId: string;
3. EmployeeName: string;
4. Experience: string;
5. Location: string;

33. }

34.

35. export default class GetSpListItemsWebPart extends BaseClientSideWebPart<IGetSpListItemsWebPartProps> {

36.

1. private \_getListData(): Promise<ISPLists> {
2. return this.context.spHttpClient.get(this.context.pageContext.web.absoluteUrl +

`/\_api/web/lists/GetByTitle('EmployeeList')/Items`, SPHttpClient.configurations.v1)

1. .then((response: Response) => {
2. debugger;
3. return response.json();

42. });

43. } 44.

45. private \_renderListAsync(): void { 46.

1. if (Environment.type === EnvironmentType.Local) {
2. this.\_getMockListData().then((response) => {
3. this.\_renderList(response.value); 50. });

51. }

1. else {
2. this.\_getListData()
3. .then((response) => {
4. this.\_renderList(response.value); 56. });

57. }

58. } 59.

1. private \_renderList(items: ISPList[]): void {
2. let html: string = '<table class="TFtable" border=1 width=100% style="border- collapse: collapse;">';
3. html +=

`<th>EmployeeId</th><th>EmployeeName</th><th>Experience</th><th>Location</th>`;

1. items.forEach((item: ISPList) => {
2. html += `
3. <tr>
4. <td>${item.EmployeeId}</td>
5. <td>${item.EmployeeName}</td>
6. <td>${item.Experience}</td>
7. <td>${item.Location}</td>

70. </tr>

71. `;

72. });

1. html += `</table>`;
2. const listContainer: Element = this.domElement.querySelector('#spListContainer');
3. listContainer.innerHTML = html;

76. } 77.

1. public render(): void {
2. this.domElement.innerHTML = `
3. <div class="${styles.helloWorld}">
4. <div class="${styles.container}">
5. <div class="ms-Grid-row ms-bgColor-themeDark ms-fontColor-white ${styles.row}">
6. <div class="ms-Grid-col ms-u-lg10 ms-u-xl8 ms-u-xlPush2 ms-u-lgPush1">
7. <span class="ms-font-xl ms-fontColor-white" style="font-size:28px">Welcome to SharePoint Framework Development</span>
8. <p class="ms-font-l ms-fontColor-white" style="text-align: center">Demo : Retrieve Employee Data from SharePoint List</p>
9. </div>
10. </div>
11. <div class="ms-Grid-row ms-bgColor-themeDark ms-fontColor-white ${styles.row}">
12. <div style="background-color:Black;color:white;text-align: center;font-weight: bold;font-size:18px;">Employee Details</div>
13. <br>
14. <div id="spListContainer" />
15. </div>
16. </div> 95. </div>`;

96. this.\_renderListAsync();

97. } 98.

1. private \_getMockListData(): Promise<ISPLists> {
2. return MockHttpClient.get(this.context.pageContext.web.absoluteUrl).then(()

=> {

1. const listData: ISPLists = {
2. value:

103. [

1. { EmployeeId: 'E123', EmployeeName: 'John', Experience: 'SharePoint',Location: 'India' },
2. { EmployeeId: 'E567', EmployeeName: 'Martin', Experience: '.NET',Location: 'Qatar' },
3. { EmployeeId: 'E367', EmployeeName: 'Luke', Experience: 'JAVA',Location: 'UK' }

107. ]

108. };

1. return listData;
2. }) as Promise<ISPLists>;

111. }

112.

1. protected get dataVersion(): Version {
2. return Version.parse('1.0');

115. }

116.

1. protected getPropertyPaneConfiguration(): IPropertyPaneConfiguration {
2. return {
3. pages: [

120. {

1. header: {
2. description: strings.PropertyPaneDescription

123. },

124. groups: [

125. {

1. groupName: strings.BasicGroupName,
2. groupFields: [
3. PropertyPaneTextField('description', {
4. label: strings.DescriptionFieldLabel

130. })

131. ]

132. }

133. ]

134. }

135. ]

136. };

137. }

138. }

### Mock HTTP Client Content

The mock http client content used to test in the local workbench is as follows:

1. import { ISPList } from './GetSpListItemsWebPart'; 2.

1. export default class MockHttpClient {
2. private static \_items: ISPList[] = [{ EmployeeId: 'E123', EmployeeName: 'John', Experience: 'SharePoint',Location: 'India' },];
3. public static get(restUrl: string, options?: any): Promise<ISPList[]> {
4. return new Promise<ISPList[]>((resolve) => {
5. resolve(MockHttpClient.\_items); 8. });

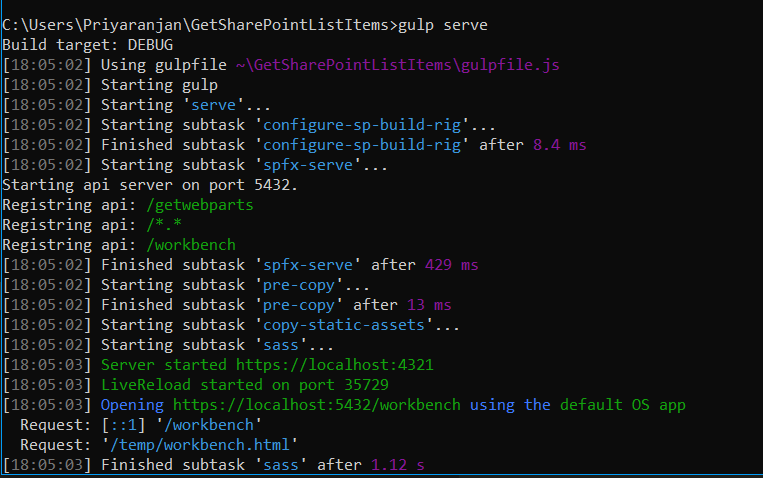
9. }

10. }

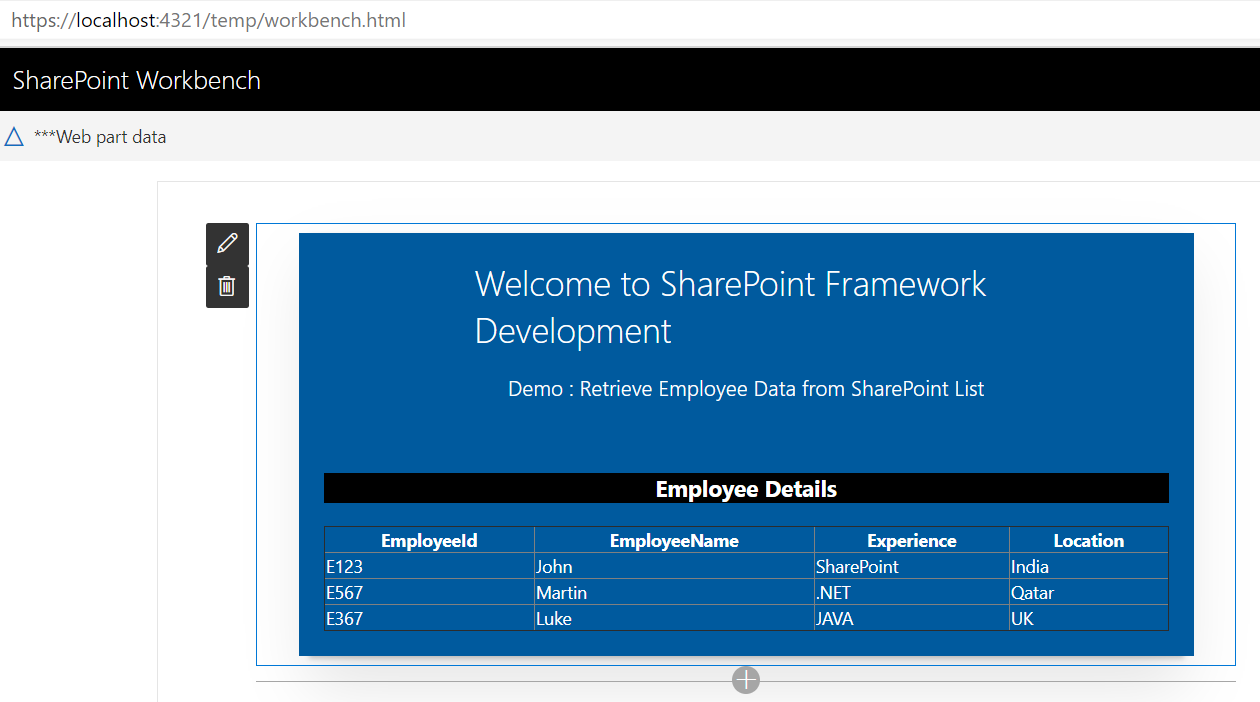
### Test the Web part in local SharePoint Workbench

Now, we can see the output generated in the local SharePoint Workbench by running *gulp serve*

command.



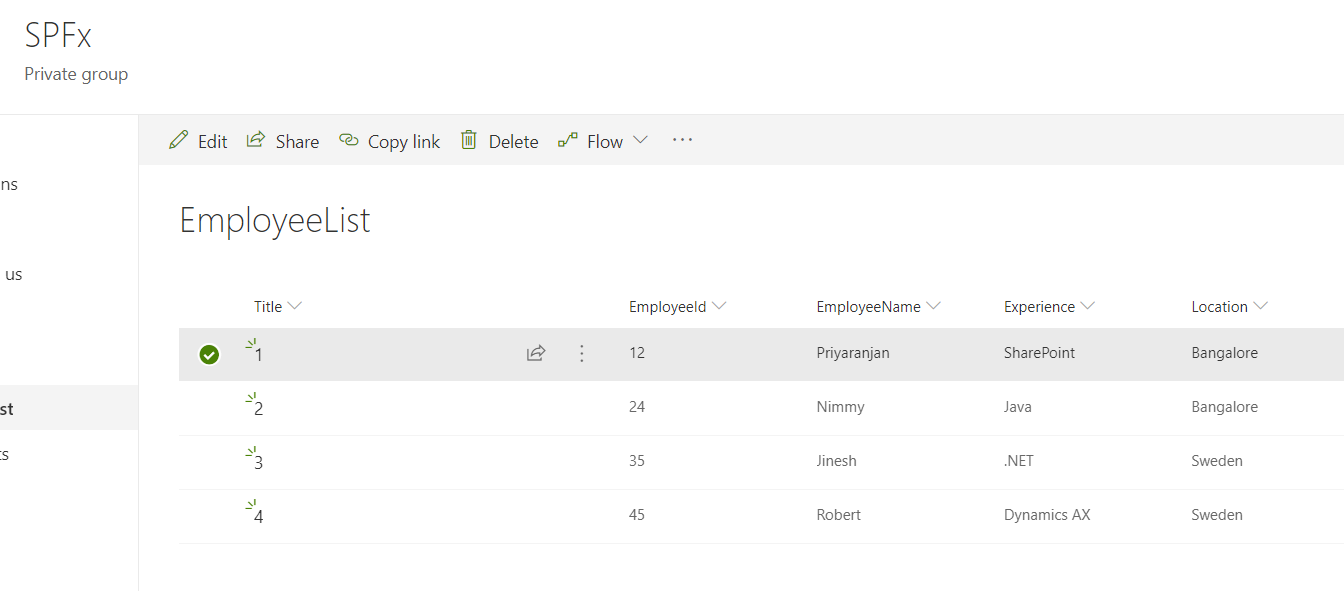
Since the environment is local, the mock data has been used to generate the table, as shown below.



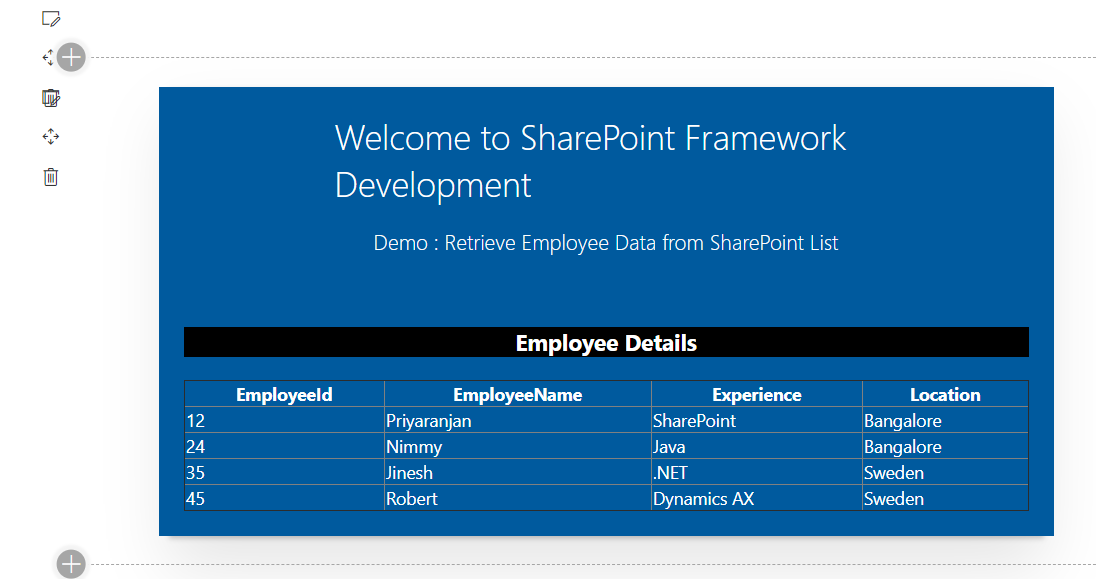
Thus, we have successfully tested the client Web part locally.

### Test the Web part in SharePoint Online

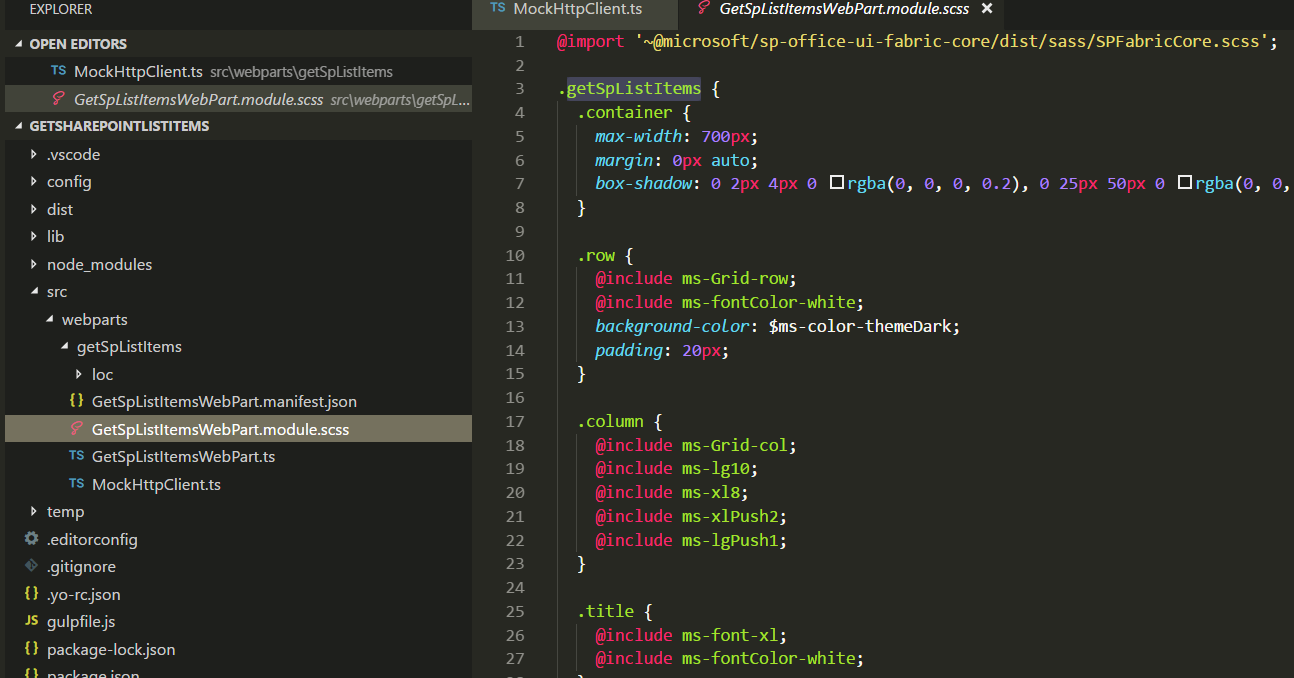
Now, let’s test the Web part in SharePoint Workbench available in SharePoint Online. This time, the 'EnvironmentType' check will evaluate to SharePoint and REST API endpoint method will be called to retrieve the list items from SharePoint list. SharePoint Online list - EmployeesList , to which we are trying to connect, using REST API is given below.



Once we have login in to SharePoint Online, we can invoke the workbench by appending the text ‘\_layouts/15/workbench.aspx’ to SharePoint Online URL. As we can see below, the items have been successfully retrieved, using REST API and the data has been built into HTML table in the client Web part.



We can further modify the CSS by making changes in the ‘GetSpListItems.module.scss’ file.



The Type Script solution file has been zipped and uploaded [here](https://gallery.technet.microsoft.com/Retrieve-SharePoint-List-24805125) . Feel free to work on it.

# Summary

Thus we saw the various building blocks and paradigms of Typescript programming and how we can carry it over to SharePoint Framework. This E Book would be updated on a regular basis with more samples.