TypeScript L-47

TypeScript

FAQ: What are the issues with JavaScript?
Ans:
- It is not strongly typed.
- It is not implicitly strictly typed.
- It is not an OOP language.
- Limited extensibility and code level security.

- Typescript is strictly typed version of JavaScript.
- It is strongly typed.
- It is an OOP language.
- Typescript is built with typescript.
- It supports low level features.
- It can directly interact with hardware services.
- It uses less memory and faster.
- Typescript can build large scale applications.
- "Anders Hejlberg" of Microsoft is the architect of TypeScript language.
  [also known for his contribution towards a language C#]
- Typescript is trans compiled into JavaScript.

Typescript Architecture:
1. Core Compiler

- core.ts	It sets up the environment for running typescript program.
- program.ts	It checks the program structure.
- scanner.ts	It is responsible for handling input.
- emitter.ts	It is responsible for handling output.
- parser.ts	It is responsible for type conversion.
- checker.ts	It is responsible for verifying the types. [data types]

2. Standalone Compiler
   - tsc.ts It is responsible for trans compiling the Typescript code into JavaScript.

3. Language Service
   - services.ts It is responsible for managing a business layer for typescript language. It provides a library of functions and values for typescript.

4. Typescript Server
   - server.ts It is responsible for hosting your application and handling request & response.
     Typescript programs are compiled and processed on server.

5. VS Shim
   - shims.ts It is responsible for making typescript cross platform.
     It generates manage code.

6. Managed Language Service	It refers to the managed library. A managed library is cross platform library.

Setup Environment for Typescript:

1. Install typescript for your PC

```
C:\>npm install -g typescript
```

2. Check the version of types

```
C:\> tsc  -v       [ latest is 5.8.2 ]
```

3. Create a new folder for typescript

```
D:\typescript
```

4. Open in folder in Visual studio code

5. Open terminal and run the following commands

```
> npm  init  -y           [ package.json ]
> tsc  -init            [ tsconfig.json ]   [tslint.json - obsolete]
```

6. Add a new typescript file into project

```
  index.ts

console.log("Welcome to TypeScript");

> tsc  index.ts           [ generates  index.js ]
> node index.js           [ runs index.js program]
```

TypeScript Language

Session-48 (TypeScript Data Types)

What is TypeScript?
Features of TypeScript
TypeScript Architecture
Setup TypeScript
What is tsconfig.json?
tsc => Typescript trans compiler
node compiler

```
> tsc index.ts
> node index.js
```

TypeScript Language

1. Variables & Data Types

- Variables are declared and initialized by using var, let, const.
- Variable naming rules are same as in JavaScript.
- TypeScript is a strongly typed language, hence every variable requires a data type to configure.

Syntax:
```
let  variableName : datatype;
```

- If data is not defined then the default type is "any".
- The type "any" is root for all data types.

Syntax:
```
let  age;        age:any;
```

- The data type for variable can be JavaScript
  a) Primitive Type
  b) Non Primitive Type

- JavaScript Primitive Data Types

  a) number
  b) boolean
  c) string
  d) null
  e) undefined
  f) Symbol
  g) bigint

Syntax:
```
let  username:string = "John";
let  age:number = 23;
let  subscribed:boolean = true;
etc..
```

- TypeScript supports "Type Inference". The data type of variable can be configured according to the value initialized.

Syntax:
```
let age = 22;          // age:number
age = "A";        // invalid
```

- TypeScript supports union of types. It allows to configure multiple types for a variable.

Syntax:
```
  let  x : number | string;
x = 10;
x = "A";
x = true;        // invalid
```

Ex:
```
let username: string | null = prompt("Enter Name");        // prompt returns string
                                                             or null.
```

TypeScript Non Primitive Types:

1. Array Type
- Typescript array can be configured for similar data types.
- It can also handle various types like JavaScript.

Syntax:
```
    let  sales:number[]  = new Array();
    let  names:string[]  =  [ ];
    let  values:any[]   = [ ];
```

- Array() constructor will not allow to initialize various data types even when the type is configured as "any". It uses the first value type as data type.
- Array() constructor will allow to assign various data types when data type any.

Syntax:
```
    let  values:any[]  = new Array(10, "A");        // invalid only numbers allowed

    let values:any[] = new Array();
    values[0] = 10;        // valid
    values[1] = "A";        // valid
    values[2] = true;        // valid
```

- Array meta character "[ ]" allows initialization and assignment of various types when data type is configured as "any".
- If a collection allows initialization and assignment of various types then its known as "Tuple".

```
     let  values:any[] = [ ];
```

- Array supports union of types only for assignment not for initialization.

Syntax:
```
     let  values:string[] | number[] = [ ];
     values[0] = 10;
     values[1] = "A";
     values[2] = 20;
```

Syntax:
```
     let  values:string[] | number[] = [10, "A" ];        // invalid
```

- All array methods are same as in JavaScript.

Ex:
index.ts

```
let values:string[]|number[] = [];

values[0] = 10;
values[1] = "A";
values[2] = 20;
values[3] = "B";

values.map(value=>{
   console.log(value);
})
```

```
> tsc index.ts
> node index.js
```

2. Object Type
- JavaScript object is schema less.
- TypeScript object is a structured object.
- It can set rules for configuring values in the reference of keys.
- Rules are defined as an object.

Syntax:
```
   let  obj : { rules } = {  data } ;
```

- Rules refer to key and value.
- Data refer to key and value.

Syntax:
```
    let  obj : {key:datatype}  = { key:value }
```

- Rules are considered as a contract, and you have to implement exactly the same that is defined in contract.
- You can't add new keys or you can't miss existing keys.
- Contract allows to configure optional keys by using null reference character "?".

Syntax:
```
    let obj : { key?:datatype } = { } ;
```

Ex:

index.ts

```
let product:{Name:string, Price:number, Rating?:number} = {
   Name: "TV",
   Price: 50000.33,
}
product.Rating = 4.5;
if(product.Rating){
   console.log(`Name=${product.Name}\nPrice=${product.Price}\nRating=${product.Rating}`);
} else {
   console.log(`Name=${product.Name}\nPrice=${product.Price}`);
```

}

Ex:
index.ts

```
let product:{Name:string, Price:number, Qty:number, Total():number, Print?():void} = {

   Name: "TV",
   Price: 56000.44,
   Qty: 2,
   Total:function(){
      return this.Qty * this.Price
   },
   Print:function(){
      console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nTotal=${this.To
tal()}`);
   }
}
product.Print();
```

- All object manipulations are same as in JavaScript.

Session-49 (TypeScript Functions)

TypeScript Language
- Variables
- Data Types
- Union of Types
- Type Inference
- Primitive & Non Primitive
- Array
- Tuple
- Object
- Optional Keys [?]

Array of Objects:
- You can configure strongly typed array with schema based objects.
- The data type is defined as object that handles array.

Syntax:
```
    let  values : { } [ ]  = [ ] ;

    {  }       => It sets rules for object
    [  ]       => It handles multiple values
```

Ex:
index.ts

```
let products:{Name:string, Price?:number}[] = [];
```

```
products = [
   {Name: 'TV', Price:5000.33},
   {Name:'Mobile', Price:1200.33}
];

products.map(product=>{
   console.log(`${product.Name} - ${product.Price}`);
})
```

Map Type:
- The class "Map" is used to configure a Map type.
- It is a generic type in typescript.
- You can make keys strongly typed for specific data type.
  [JavaScript map keys handle any type]
- You can configure any or restrict to specific.

Syntax:

```
   let  data : Map<keyType, valueType> =  new Map();

   data.get()
   data.set()
   data.has()
   data.keys()
   data.values()
   data.entries()
   data.delete()
   data.clear() etc..
```

Ex:

```
let data:Map<string, number> = new Map();
let list:Map<number, any> = new Map();
let values:Map<any, any> = new Map();

data.set("A", 1200);

console.log(data.get("A"));
data.keys()
data.entries()
data.values()
```

Date Type:
- You can configure strongly typed date values in typescript.
- All date and time functions are same as in JavaScript.

Syntax:

```
    let  now : Date   = new Date();        // loads current date & time
    let  mfd  : Date  = new Date("yy-mm-dd hrs:min:sec.milliSec");

   now.get..()
   now.set..()
```

Regular Expression Type:
- Typescript provides "RegExp" as data type for regular expression.
- It allows meta characters and quantifiers enclosed in "/ /".

Syntax:
```
    let pattern: RegExp = /\+91\d{10}/;
```

- Value is verified with regular expression by using "match()" method.
- The value type must be any type to verify with regular expression.

Note: Statements and Operators are same as in JavaScript.

1. Operators
   - Arithmetic
   - Logical
   - Comparison
   - Assignment
   - Bitwise
   - Special etc.

2. Statements
   Selection
      if, else, switch, case, default
   Iteration
      for..in, for..of
   Looping
      for, while, do while
   Jump
      break, continue, return
   Exception handling
      try, catch, throw, finally

Typescript Functions

- You can configure a function with declaration or expression.
- Typescript function declaration comprises of
   a) parameter data type
   b) function with return data type or void

Syntax:
```
     function  Name(param : datatype) : datatype | void
     {
     }
```

Ex:
index.js

```
function Addition(a:number, b:number):number
{
     return a + b;
}

function Print():void
{
    console.log(`Addition=${Addition(50,20)}`);
}
Print();
```

- Typescript function can have optional parameters.
- Optional parameter must be last parameter.
- A required parameter can't follow an optional parameter.

Syntax:

```
     function Name(param1:type,  param2?:type)        // valid
     {
     }

     function Name(param1?type: param2:type)           // invalid
     {
     }
```

- Optional parameters are verified by using "undefined" type.

Ex:

```
index.ts
function Details(id:number, name:string, price?:number):void
{
     if(price) {
         console.log(`id=${id}\nname=${name}\nprice=${price}`);
     } else {
        console.log(`id=${id}\nname=${name}\n`);
     }
}
Details(1, "TV");
Details(2, "Mobile", 5000.44);
```

- A function can configure rest parameters.
- Rest parameter must be defined as array type.
- It can be strongly typed array with specific type of arguments or any type.

Syntax:

```
      function  Name(...params: number[])
      {
      }
      Name(10, 20, 50);

      function Name(...params: any[])
      {
      }
```

```
     Name (1, "A", true);
```

Ex:
index.ts

```
function Details(...product:any[]):void
{
     let [id, name, price] = product;

     if(price) {
         console.log(`id=${id}\nname=${name}\nprice=${price}`);
     } else {
        console.log(`id=${id}\nname=${name}\n`);
     }
}
Details(1, "TV");
Details(2, "Mobile", 5000.44);
```

Type Script Promise:
- Promise function uses a generic type.
- It allows to restrict specific data typed on resolve.
- However the reject function is always any type.

Syntax:

```
    let  fetch : Promise<datatype> = new Promise(function(resolve, reject){

        resolve(datatype);
        reject(any_type);

    })

    fetch.then((data)=>{ }).catch((error)=>{ }).finally(()=>{ })
```

Ex:
index.ts

```
let FetchData:Promise<{Name:string, Price:number}> = new Promise(function(resolve, reject){

        let url = "https://server.com&quot;;

        if(url==="https://fakestore.com&quot;){
             resolve({Name:'TV', Price:50000});
        } else {
             reject('Invalid URL - API Not found');
        }
})

FetchData.then(function(data){
    console.log(data);
})
```

```
.catch(function(error){
   console.log(error);
})
```

Type Script OOP

Session-50 (TypeScript Contracts)

Typescript Language
- Variables
- Data Types
- Operators
- Statements
- Functions

Typescript OOP

Contracts in OOP:
- A contract defines rules for designing any component.
- It enables reusability, separation and easy extensibility.
- In OOP contract as designed as "interface".

Syntax:
```
      interface  IName
      {
           // rules
      }
```

- Contract must contain only declaration not implementation.
- A contract can define rules for both property and method.

Syntax:
```
      interface  IName
      {
         property: datatype;
         method(): datatype | void;
      }
```

- A contract can have optional rules defined using null reference character [?].

Syntax:
```
      interface IName
      {
         property?: datatype;
         method?(): datatype | void;
      }
```

- Optional rules are required to define goals for any contract.

Note: Objective is time bound and mandatory to implement.
    Goal is not time bound and optional to implement.

- A contract can have read-only rules.
- Every rule can be assigned with a new value or functionality after initialization.
- You can prevent assignment by configuring the rule as "readonly".

Syntax:
```
    interface  IName
    {
      readonly  property: datatype;
    }
```

- You can extend a contract by adding new rules.
- Extensibility is achieved by using inheritance.
- A contract "extends" another contract.
- It allow to implement multiple contracts. [multiple inheritance]
- It is possible as interface will not have a constructor.
- Interface will not allow to create an instance.

Syntax:
```
     interface   IName extends  Contract1, Contract2
     {
     }
```

Ex:
```
interface ICategory
{
   CategoryName?:string;
}

interface IRating {
   Rating:number;
}

interface IProduct extends ICategory, IRating
{
   Name:string;
   readonly Price:number;
   Qty:number;
   Total():number;
   Print?():void;
}


let tv:IProduct = {
   Name : "Samsung TV",
   Price: 40000.44,
   Qty: 2,
   CategoryName: "Electronics",
   Rating: 4.2,
   Total() {
      return this.Qty * this.Price
```

```
    },
    Print(){
        console.log(`Name=${this.Name}\nPrice=${this.Price}\nTotal=${this.Total()}\nCategory=
${this.CategoryName}\nRating=${this.Rating}`);
    }
};
tv.Print();
```

- A contract is used as type for any memory reference.
- The default type is object, you can configure as array.

Syntax:

```
    let obj : IProduct = {  };

    let obj : IProduct[] = [ { }, { } ];
```

Classes:
- Class declaration and expression is same as in JavaScript.
- It supports ES6+ static members.
- Class members are same
  - a) Property
  - b) Accessor
  - c) Method
  - d) Constructor
- Class supports access modifiers like
  - a) public
  - b) private
  - c) protected

FAQ: What is difference between static and non-static?
Ans:
 a) Static
   - It refers to continuous memory.
   - Memory allocated for first object will continue for next.
   - It uses more memory.
   - It is good for continuous operations.
   - A static member is accessible with in or outside class by using class name.

 b) Non Static | Dynamic
   - It refers to discreet memory.
   - Memory is newly allocated for object every time.
   - It is safe and uses less memory.
   - It is good for disconnected actions.
   - It is accessible with in class by using "this" keyword and outside class by using and instance of class.

POC:

```
class Demo
{
    static s = 0;
    n = 0;
    constructor(){
        Demo.s = Demo.s + 1;
        this.n = this.n + 1;
    }
    Print(){
        console.log(`s=${Demo.s} n=${this.n}`);
    }
}

let obj1 = new Demo();
obj1.Print();

let obj2 = new Demo();
obj2.Print();

let obj3 = new Demo();
obj3.Print();
```

Session-51 (TypeScript Templates)

TypeScript OOP
- Contracts / Interface
- Classes
    - Members are same as in JS
    - Static and Non Static Members

Access Modifiers:
1. public
2. private
3. protected

- public allows access from any location and by using any instance.
- private allows access with in class.
- protected allows access with in class and from derived class only by using derived class reference. [or instance]

Ex:
```
class Super
{
    public Name:string = "TV";
    private Price:number = 40000;
    protected Stock:boolean = true;
    public Print():void {
        console.log(`${this.Stock}\n${this.Name}\n${this.Price}`);
    }
}
class Derived extends Super
{
    public Print():void {
        let obj = new Derived();
        obj.Stock;  // protected
        obj.Name;   // public
    }
}

let obj = new Derived();
obj.Name;       // public
```

- Inheritance and rules are same as in JavaScript. [extends]
- A class can implement the contract.
- Contract is used to defined rules for a class.
- Class implements contract and allows to customize by adding new properties and methods.
- Class is a program template, hence it allows customization.
- Class can implement multiple contracts.

Syntax:
```
        class Name implements Contract1, Contract2
        {
        }
```

Ex:
```
interface IProduct
{
    Name:string;
    Price:number;
}
interface ICategory
{
    CategoryName:string;
}

class Product implements IProduct, ICategory
{
    public Name = "TV";
    public Price = 50000;
    public CategoryName = "Electronics";
}
```

Templates in OOP:
- A template provides pre-defined structure.
- It comprises of some features implemented and some of them need to be implemented.
- Client implements templates by customizing according to the requirements.
- Templates are mostly used for Rollouts and Implementation of secured modules in application for any business.
- Template hides its structure and provides only implementation.
- The process of hiding structure and providing only implantation to developer is known as "Abstraction".
- Typescript can create template as "Abstract Class".

Syntax:
```
    abstract class Name
```

```
   {

   }
```

- Abstract class can have members implemented and yet to implement.
- The members that require implementation are marked as "abstract".

```
   {
     public abstract property;
     public abstract method();
   }
```

- If a class have at least on abstract member then it must be marked as abstract.
- You have extend abstract class to implement the abstract member.
- You can't create instance for abstract class. But it can have a constructor.

Ex:

```
interface IProduct
{
   Name:string;
   Price:number;
   Qty:number;
   Total():number;
   Print():void;
}
abstract class ProductTemplate implements IProduct
{
   constructor(){
      console.log("Abstract class Constructor");
   }
   public Name:string = "";
   public Price:number = 0;
   public Qty:number = 1;
   public abstract Total():number;
   public abstract Print():void;
}
class Component extends ProductTemplate
{
      Name = "Samsung TV";
      Price = 50000;
      Qty = 2;
      Total(){
         return this.Qty * this.Price;
      }
      Print(){
         console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nTotal=${this.
Total()}`);
      }
}
```

```
let tv = new Component();
tv.Print();
```

Session-52 (Generics and Enum)

Contracts
- interface

Templates
- abstract class

Components
- class

Generics

- Generic is used to configure "type safe" component or property.
- It is open to handle any type of data, and makes it strongly typed once it knows the data type of value.
- Typescript allows generic
  a) method
  b) function
  c) parameter
  d) class
  e) property etc..

- Generic type is defined by using "<T>" type reference.

Syntax:
```
      function Name<T>(param: T): T
      {
         return  param | expression;
      }
```

Ex:
```
function Print<T>(param:T):void{
   console.log(`Param=${param}`);
}

Print<number>(30);
Print<string>("A");
Print<string[]>(["A","B"]);
```

Ex: Generic Class & Property

```
interface IOracle
{
   UserName:string;
   Password:string;
   Database:string;
}
interface IMongoDB
{
    URL:string;
}

class Database<T>
{
    public connectionString:T|undefined;
    public Print():void{
        for(var property in this.connectionString){
            console.log(`${property} : ${this.connectionString[property]}`);
        }
    }
}

let oracle = new Database<IOracle>();
oracle.connectionString = {UserName:'scott', Password:'tiger', Database:'stu'};
oracle.Print();

let mongodb = new Database<IMongoDB>();
mongodb.connectionString = {URL: 'mongodb://127.0.0.1:27017'};
mongodb.Print();
```

Ex:

```
interface IProduct
{
    Name:string;
    Price:number;
}
interface IEmployee
{
    FirstName:string;
    LastName:string;
    Designation:string;
}

class FetchData<T>
{
     public Response<T>(data:T){
         console.log(data);
     }
}

let product = new FetchData<IProduct>();
product.Response<IProduct>({Name: 'TV', Price:55000});
product.Response<IProduct[]>([{Name:'Mobile', Price:30000}, {Name:'Watch', Price: 13000}]);
```

- You can't use operators on generic types.
- Generics are always handled by using methods or functions.

Syntax:

```
        function Add<T>(a:T, b:T) : T
        {
          return  a + b;    // invalid
        }
```

Ex:

```
function Sum(a:any,b:any){
   return a + b;
}

function Addition<T>(a:T, b:T):T
{
   return Sum(a,b);
}

console.log(`Addition=${Addition<number>(40,20)}`);
```

Enumeration [Enum]

- It is a collection of constants.
- Enum can have numeric, string or expression as constant.
- It can't have Boolean value or expression as constant.
- Enum allows auto implementation of values if the type is "number".

Syntax:

```
   enum Values
   {
      A,              // A=0
      B=20,
      C               // C=21
   }
   Values.A;
   Values.A = 20;    // invalid
```

- Auto implementation is allows only when the previous value type is a number.
- Enum allows reverse mapping.
- It is a technique of accessing key with reference of value.

Ex:

```
enum StatusCodes
{
    Client,
    OK = 200,
    Found,
    NotFound = 404,
    Server = "Inernal Server Error",
}
console.log(`${StatusCodes.NotFound}: ${StatusCodes[404]}`);
```

Ex:

```
enum StatusCodes
{
   A = 10,
   B = 20,
   C = A + B
}
console.log(`Addition=${StatusCodes.C}`);
```

Module System

- A module system enables features like
   a) Code Separation
   b) Code Reusability
   c) Extensibility
   d) Maintainability
   e) Testability
- TypeScript uses CommonJS module system.
- A Typescript module comprises of
   a) Contracts
   b) Templates
   c) Components
   d) Functions
   e) Values

Ex:
1. Add folders
   - contracts
   - templates
   - components
   - app

2. contracts
   ProductContract.ts

```
export interface ProductContract
{
   Name:string;
   Price:number;
   Qty:number;
   Total():number;
   Print():void;
}
```

3. templates
   ProductTemplate.ts

```
import { ProductContract } from "../contracts/ProductContract";

export abstract class ProductTemplate implements ProductContract
{
     public Name:string = "";
     public Price:number = 0;
     public Qty: number = 0;
     public abstract Total():number;
     public abstract Print():void;
}
```

4. components
   ProductComponent.ts

```
import { ProductTemplate } from "../templates/ProductTemplate";

export class ProductComponent extends ProductTemplate
{
    Name = "Samsung TV";
    Price = 50000;
    Qty = 2;
    Total(){
      return this.Qty * this.Price;
    }
    Print(){
      console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nTotal=${this.To
tal()}`);
    }
}
```

5. app
   Index.ts

```
import { ProductComponent } from "../components/ProductComponent";

let tv = new ProductComponent();
tv.Print();
```

```
   D:\..\app> tsc index.ts
           > node index.js
```

Session-53 (Namespace and React with TypeScript)

Namespace
- A namespace is a collection of sub-namespaces and OOP components.
- It can have a set of contracts, templates or components.
- It is used to build and organize libraries at large scale.

Syntax:

```
        namespace  Project
        {
         namespace  Module
         {
              // contracts
          // templates
          // components
         }
        }
```

- A namespace is imported by using "///<reference />" directive.

Syntax:

```
      ///<reference  path="./folder/module" />
```

- You can alias namespace of access using fully qualified name.

Syntax: Fully Qualified

```
      Project.Module.ClassName

      let obj = new Project.Module.ClassName();
```

Syntax: Aliasing Namespace

```
      import   className = Project.Module.ClassName;

      let obj = new className();
```

- To compile the typescript library with namespace you have to use the command

```
      > tsc -outFile  index.js  index.ts

      > node index.js
```

Note: In typescript latest release the parent name space doesn't require export.
All other members must be marked as export.

Ex:
1. Add folders
   -contracts
   -templates
   -components
   -app

2. contract/ProductContract.ts

```
namespace Project
{
    export namespace Contracts
     {
          export interface IProduct
          {
             Name:string;
             Price:number;
             Qty:number;
             Total():number;
             Print():void;
          }
    }
}
```

3. template/ProductTemplate.ts

```
///<reference path="../contracts/ProductContract.ts" />

import ProductContract = Project.Contracts.IProduct;

namespace Project
{
    export namespace Templates
    {
        export abstract class ProductTemplate implements ProductContract
        {
            public Name:string = "";
            public Price:number = 0;
            public Qty:number = 0;
            public abstract Total():number;
            public abstract Print():void;
        }
    }
}
```

4. components/ProductComponent.ts

```
///<reference path="../templates/ProductTemplate.ts" />

import ProductTemplate =  Project.Templates.ProductTemplate;

namespace Project
{
     export namespace Components
     {
         export class ProductComponent extends ProductTemplate
         {
             Name = "Samsung TV";
             Price = 45000;
             Qty = 2;
             Total(){
                return this.Qty * this.Price;
             }
             Print(){
                console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nTotal=${
this.Total()}`);
             }
         }
     }
}
```

5. app/index.ts

```
///<reference path="../components/ProductComponent.ts" />

import ProductComponent = Project.Components.ProductComponent;

let tv = new ProductComponent();
tv.Print();
```

6. Compile and Run

```
   D:..\app> tsc  -outFile  index.js  index.ts

            > node index.js
```

React Application with TypeScript

- You can use various bundling tools for creating react app.
    a) Webpack
    b) Vite
    c) Parcel
- Vite is a build tool that provides various features for developer to
    a) build
    b) debug
    c) test
    d) deploy

Creating a new Project using Vite and TypeScript for React:

1. Make sure that your device is installed with Node 18+ version

2. Open your PC location in command prompt

```
    D:\>npm  create  vite@latest   app-name  -- --template  react-ts    [Typescript]
    D:\>npm  create  vite@latest   app-name  -- --template  react    [JavaScript]
```

3. A project folder is created at specified location

4. Change into the folder and run the command

```
    D:\app-name> npm  install
```

   - It will install react, react-dom, typescript dependencies.

5. Run your project

```
   D:\app-name> npm run dev
```

   http://localhost:5173

React Vite App File System:

```
node_modules        : comprises of library files installed using npm.
public              : comprises of static resources
src                 : comprises of dynamic resources
.gitignore          : configuration to ignore folders while publishing to GIT
eslint.config.js    : It is JavaScript language analysis tool
index.html          : startup page
package.json
package.lock.json
README.md
tsconfig.json       : typescript configuration file
tsconfig.app.json   : typescript configuration for current app
tsconfig.node.json  : node JS migrations
vite.config.ts      : vite configuration for app
                      [It configures react for your app]
```

Note: The index.js is now replaced with "main.tsx" in "src" folder.
index.html is importing main.tsx

Session-54 (TypeScript App)

What's new with React TypeScript:

1. Component is a function that returns JSX.Element

   - TypeScript supports type inference, hence the return type is not mandatory to define.

```
Syntax:
export function Login() : JSX.Element
{
    return (
       <>
          JSX
       </>
       );
 }
```

2. JSX Rules are same.

3. Component file must have extension .tsx.

   Login.tsx

Note: All library files must have extension ".ts".
- Contracts
- Templates

Component & Hooks are defined as ".tsx".

4. Component state is generic type.

```
const [get, set] = useState<T>();

const [categories, setCategories] = useState<string[]>([ '  ', '  ']);
```

You can use a contract for configuring data type.

```
const [product, setProduct] = useState<IProduct>();

interface IProduct
{
  Id:number;
  Name:string;
}
```

5. Data Binding, Style Binding, Class Binding & Event Binding all are same.

6. All hooks are same.

7. Axios, Formik, Yup, Routing, cookies etc. all are same.

8. Controlled components uses Props type is an any type object.

Syntax:
```
export function Navbar(props:any)
{
}
```

9. Setup Bootstrap or MUI for project is same. MUI components are same.

10. Import bootstrap, cookie provider and other global provides in "main.tsx".

Note: The state configure for handling data must be strongly typed.
It is default nullable, if value is not initialized.

Syntax: If value is not initialized
```
const [categories, setCategories] = useState<string[]>();

categories?.map(category=> <li> </li>)
```

Syntax: If value is initialized

```
const [categories, setCategories] = useState<string[]>([ ' ' ]);

categories.map(category=> <li></li>)
```

Ex: Fakestore API

1. Add contracts folder into "src"

2. Add a new file "fakestore-contract.ts"

```
export interface FakestoreContract
{
   id:number;
   title:string;
   description:string;
   image:string;
   price:number;
   rating: {rate:number, count:number}
}
```

3. demo.tsx

```
import { useEffect, useState } from "react";
import { FakestoreContract } from "../contracts/fakestore-contract";
import axios from "axios";


export function Demo(){


   const [categories, setCategories] = useState<string[]>();
   const [products, setProducts] = useState<FakestoreContract[]>();

   function LoadCategories(){
      axios.get(`https://fakestoreapi.com/products/categories`)
      .then(response => {
         setCategories(response.data);
      })
   }
   function LoadProducts(){
      axios.get(`https://fakestoreapi.com/products`)
      .then(response=>{
         setProducts(response.data);
      })
   }

   useEffect(()=>{
      LoadCategories();
      LoadProducts();
   },[])

   return(
      <div className="container-fluid">
         <header>
            <h3 className="text-center">Fakestore</h3>
         </header>
```

```
            <section className="row mt-4">
                <nav className="col-2">
                    <label className="form-label">Select Category</label>
                    <select className="form-select">
                        {
                            categories?.map(category=><option key={category}>{category}</option>)
                        }
                    </select>
                </nav>
                <main className="col-10 d-flex flex-wrap overflow-auto" style={{height:'400px'}}>
                     {
                        products?.map(product=>
                            <div key={product.id} className="card p-1 m-2" style={{width:'200px'}}>
                                <img src={product.image} className="card-img-top" height='120' />
                                <div className="card-header" style={{height:'100px'}}>
                                    {product.title}
                                </div>
                            </div>
                        )
                     }
                </main>
            </section>
        </div>
    )
}
```

MERN Stack Application
[Project]

- Video Library Application

    Admin Module

    * Admin can login
    * Admin can add videos
    * Edit videos
    * Delete videos

    User Module

    * User can register
    * User can login
    * User can browse and view videos
    * User can search videos by title, category etc.
    * User can save videos to watch later. [My List]

- Libraries and Frameworks

MongoDB	: for database
Express JS	: for middleware
Node JS	: for server side app
React	: for UI - front end

React Router	: for routing
Axios	: for API communication
Formik	: for form
Yup	: for validation
React Cookies	: for user state
Redux Toolkit	: for Videos watch later [application state]

Setup Database for Application

- MongoDB database
- It is an Non-SQL database [no-SQL]
- It uses BSON type data types.
- It is similar to JSON.
- It provides simple methods for handling CRUD operations.

1. Install MongoDB community server on your device.

2. Select "MongoDB compass" tool while installing MongoDB server.

3. Compass is a GUI tool that provides an UI to handle database on server.

   https://www.mongodb.com/try/download/community

4. After installing MongoDB start its server

   - Go to "services.msc" from your programs
   - Right Click on "MongoDB Server" and select start.

5. Open MongoDB compass from programs

6. Connect to Server using following connection string

   mongodb://127.0.0.1:27017

7. After connecting you will find default databases

   a) admin
   b) config
   c) local

MongoDB Terminology:

Oracle, MySQL	MongoDB
Database	Database
Table	Collection
Record / Row	Document
Field / Column	Field / Key

Join   Embedded Document

find()
insertOne()
insertMany()
updateOne()
updateMany()
deleteOne()
deleteMany()

Session-55 (MongoDB and Server App)

Video Library Project - MERN Stack
Setup Database - MongoDB

CRUD Operations in MongoDB:
- Database
- Collection [Table]
- Document [Record]
- Field

1. Open MongoDB Compass
2. Connect with MongoDB server
   mongodb://127.0.0.1:27017
3. Select connection
4. Open MongoDB Shell. It is a CLI tool for handling database.

Commands:
1. To create a new database or to start using existing database

```
> use  databaseName

> use  demodb        // creates a new database if doesn't exist
```

Note: Database will be displayed in list only when it is having collections.

2. To create a new collection [table]

```
>db.createCollection("name", { attributes })

>db.createCollection("products")
```

3. To view databases and collections

```
> show dbs           // to view database
> show collections       // to view collections [tables]
```

4. Add data into collection [table]

a) insertOne()
b) insertMany()

```
> db.collectionName.insertOne({key:value})
> db.collectionName.insertMany([{key:value}, {key:value},..])
```

All JavaScript data types are used for MongoDB

Ex:
```
> db.products.insertMany([{Id:1, Name:"TV", Price:23000, Rating:{Rate:4.2, Count:500}}, {Id:2, Name:"Mobile", Price:12000, Rating:{Rate:4.1, Count:230}}])
```

5. Read data from collection
a) find()
b) findOne()

Syntax:
```
> db.collectionName.find({query})

> db.products.find({})          // returns all documents
> db.products.find({id:3})
```

Operators:

$gt	greater than
$gte	greater than or equal
$lt	less than
$lte	less than or equal
$eq	equal
$ne	not equal
$or	OR
$and	AND

```
> db.products.find({Price:{$gte:5000}})
> db.products.find({'Rating.Rate': {$gt:4}})
> db.products.find({$and:[{Category:"Electronics"}, {Price:{$gte:5000}}]})
```

6. Update Document

a) updateOne()
b) updateMany()

```
> db.collectionName.updateOne({findQuery}, {updateQuery})
```

Operators:

$set	update data into specified field
$unset	removes a field
$rename	changes field name

```
> db.products.updateOne({Id:1}, {$set:{Price:25000, Name:"Samsung TV"}})
```

7. Delete Document

    a) deleteOne()
    b) deleteMany()

```
> db.products.deleteOne({findQuery})

> db.products.deleteMany({Category:"Electronics"})
> db.products.deleteOne({Id:3})
```

Zoom Meeting ID: 91690775166
PassCode   : 112233

Timing   9:30 AM to 6:30 PM

Server Side Application
[Node & Express JS]

- Node is a JavaScript runtime used to build web applications, scripts, command line tools etc.
- Express JS is a middleware framework for Node JS.
- Middleware enables communication between client-server-database.
- You can create API using node & expression.

1. Create a new folder on your PC

```
D:\server-app
```

2. Setup package JSON

```
>npm init -y
```

3. Install the following libraries

```
> npm  install  express  --save
> npm  install  cors  --save
```

4. Add a new file "server.js"

```
const express = require("express");

const app = express();

app.get("/", (req, res)=>{
    res.send("Welcome to API");
    res.end();
});
```

```
app.get("/products", (req, res)=>{
    res.send([{Name:"TV"},{Name:"Mobile"}]);
    res.end();
});

app.listen(5050);
console.log(`Server Started http://127.0.0.1:5050`);
```

5. Run

```
   > node server.js
```

Session-56 (Video Library Project - Data & API)

Create Database and Collection for Project

1. Create a new database on MongoDB
      "video-tutorial"

2. Add collections

      1. admin [collection]

```
      {
        admin_id : string,
        password : string
      }
```

      2. users [collection]

```
      {
        userid: string,
        username: string,
        password: string,
        email:string
      }
```

      3. videos [collection]

```
      {
        video_id : number,
        title: string,
        description: string,
        url: string,
        likes: number,
        views: number,
        dislikes: number,
        category_id: number [FK]
      }
```

      4. categories [collection]

```
 {
  category_id: number, [PK]
  category_name: string
 }
```

Create API to handle request [End Points]

```
GET     /admin              get admin users
GET     /users              get all users
GET     /videos             get all videos
GET     /videos/1           get specific video by id
GET     /categories         get all categories

POST    /register-user      add new user into database
POST    /add-video          add new video into database
PUT     /edit-video/1       update and save video details
DELETE  /delete-video/1     delete video by ID.
```

1. Go to your server app [add server folder into react-typescript-app]

2. Install following libraries

```
> npm install  express  mongodb  cors  --save
```

```
express      : It is a middleware for configuring API end points
mongodb      : It is a drivers library to connect with MongoDB database
cors         : Cross Origin Resource Sharing for handling request like
               GET, POST, PUT, DELETE. [restrictions]
```

3. Add a new file "api.cjs" into server folder

```
// import libraries

const cors = require("cors");
const express = require("express");
const mongoClient = require("mongodb").MongoClient;

// Create connection string and app

const conString = "mongodb://127.0.0.1:27017";

const app = express();
app.use(cors());
app.use(express.urlencoded({extended:true}));
app.use(express.json());

// Create API end points

app.get('/admin',(req, res)=>{
    mongoClient.connect(conString).then(clientObject=>{

        var database = clientObject.db("video-tutorial");

        database.collection("admin").find({}).toArray().then(documents=>{
              res.send(documents);
              res.end();
        });
    });
});

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

    mongoClient.connect(conString).then(clientObject=>{

        var database = clientObject.db("video-tutorial");

        database.collection("users").find({}).toArray().then(documents=>{
              res.send(documents);
              res.end();
        });
    });
});

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

    mongoClient.connect(conString).then(clientObject=>{

        var database = clientObject.db("video-tutorial");

        database.collection("videos").find({}).toArray().then(documents=>{
              res.send(documents);
              res.end();
        });
    });
});

app.get('/videos/:id',(req, res)=>{

    var id = parseInt(req.params.id);

    mongoClient.connect(conString).then(clientObject=>{

        var database = clientObject.db("video-tutorial");

        database.collection("videos").findOne({video_id:id}).then(document=>{
              res.send(document);
              res.end();
        });
    });
```

```
});

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

    mongoClient.connect(conString).then(clientObject=>{

        var database = clientObject.db("video-tutorial");

        database.collection("categories").find({}).toArray().then(documents=>{
              res.send(documents);
              res.end();
        });
    });
});

app.post('/register-user', (req, res)=>{

        var user = {
            userid: req.body.userid,
            username: req.body.username,
            password: req.body.password,
            email: req.body.email
        };

        mongoClient.connect(conString).then(clientObject=>{

            var database = clientObject.db("video-tutorial");

            database.collection("users").insertOne(user).then(()=>{
                console.log('User Registered');
                res.send();
            });
        });
});


app.post('/add-video', (req, res)=>{

    var video = {
        video_id : parseInt(req.body.video_id),
        title: req.body.title,
        description: req.body.description,
        url: req.body.url,
        likes: parseInt(req.body.likes),
        dislikes: parseInt(req.body.dislikes),
        views: parseInt(req.body.views),
        category_id: parseInt(req.body.category_id)
    };

    mongoClient.connect(conString).then(clientObject=>{
        var database = clientObject.db("video-tutorial");

        database.collection("videos").insertOne(video).then(()=>{
            console.log('Video Added');
            res.send();
        });
    });
});

app.put('/edit-video/:id', (req, res)=>{

    var id = parseInt(req.params.id);

    var video = {
        video_id : parseInt(req.body.video_id),
        title: req.body.title,
        description: req.body.description,
        url: req.body.url,
        likes: parseInt(req.body.likes),
        dislikes: parseInt(req.body.dislikes),
        views: parseInt(req.body.views),
        category_id: parseInt(req.body.category_id)
    };

    mongoClient.connect(conString).then(clientObject=>{

        var database = clientObject.db("video-tutorial");

        database.collection("videos").updateOne({video_id:id},{$set: video}).then(()=>{
            console.log('Video Updated');
            res.send();
        });
    });
});

app.delete('/delete-video/:id', (req, res)=>{

    var id = parseInt(req.params.id);

    mongoClient.connect(conString).then(clientObject=>{

        var database = clientObject.db("video-tutorial");

        database.collection("videos").deleteOne({video_id:id}).then(()=>{
            console.log('Video Deleted');
            res.send();
        });
    });
});
```

```
app.listen(4040);
console.log(`Server Started http://127.0.0.1:4040`);
```

4. Go to package.json

```
   scripts: {

       "api"  :  "node  ./server/api.cjs"

   }
```

 5. From terminal you can run command

```
   >npm run api
```

Build UI in React - Using Typescript as Language:

Session-57 (Video Project - UI)

Creating Database for Project
   - MongoDB
Creating API and Configure End Points
   - Node & Express JS

Creating UI with React:

1. Install required libraries for React app

```
   > npm install  bootstrap bootstrap-icons  formik yup axios  react-cookie --save
   > npm install @mui/material @emotion/react @emotion/styled  --save
   > npm install  react-router-dom --save
```

2. Go to "main.tsx" and import Cookies Provider

3. Set "App" component as startup component.

```
     import {  CookiesProvider }  from  'react-cookie';

     <CookiesProvider>
         <App />
     </CookiesProvider>
```

4. Import bootstrap & icons CSS into main.tsx

5. Add contracts folder into "src" and setup all contracts required to connect with database in backend.

```
        contracts/admin-contract.ts
```

contracts/user-contract.ts
contracts/video-contract.ts
contracts/categories-contract.ts

6. Add components folder into src with following component files

    user-login.tsx
    user-dash.tsx
    admin-login.tsx
    admin-dash.tsx
    admin-add-video.tsx
    admin-edit-video.tsx
    admin-delete-video.tsx
    register-user.tsx
    video-home.tsx

Session-58 (React Redux) + Video Project.Zip

Functionalities to Implement
- Likes, dislikes counter must increase and store in database
- User must able to view the video separately in a new route.
- It must increase the views count.
- User must able to search by category or title.
- User can register a new account.
- Setup validation while adding, editing videos.

React Redux

- Redux is a JavaScript library.
- It is used to configure and maintain global application memory for JavaScript based applications.
- It is a large scale version of application memory when compared to useReducer in React.
- It is predictable and debuggable.
- It provides a complete toolkit for developers to manage application state.
- Redux can be used with angular, react, vue and all JavaScript apps.

Redux Components:
  a) Store
  b) State
  c) Reducer

- Store is the location where data is kept.
- State can access data from store and update to UI.
- Reducer comprises of actions required to update data in store.

Setup Redux for Project

1. Install Redux toolkit with React support

```
> npm install @reduxjs/toolkit   react-redux  --save
```

2. Redux toolkit provides
   a) Slicer
   b) Store
   c) Reducer
   d) Initial State

3. Create a new slicer

   - Slicer configure the initial data to store in global memory.
   - It initializes the global memory.
   - It uses initial state.
   - You can create by using "createSlice()" method.
   - It also defines the actions to perform
   - Actions are required to update the data in global memory.

Syntax:

```
        video-slicer.tsx

      let initialState = {
          videos : [ ],
          videosCount: 0
       }

     const  videoSlice = createSlice({
        name: 'video',
            initialState,
        reducers: {
            addToList: (payload)=>{ videos.push(payload) }
        }
      })

    export  videoSlice.actions;
```

4. Configure a store from Redux toolkit

   - It requires configureStore() method
   - It can create a store at application level
   - It can get data from your reducer and update into store.
   - Store is provided Global so that you can access from any component.

Syntax:

```
     store.tsx

     import { configureStore }  from  "@redux/toolkit";

    export function configureStore(){
        // specify the reducers.
    }
```

5. Go to main.tsx and set provider for store.

Syntax:
```
<Provider  store={ store } >
   <App />
</Provider>
```

Session-59 (Redux Toolkit)

Redux in Video Library Project

1. Install Redux with React support into project

```
> npm install  @reduxjs/toolkit   react-redux   --save
```

2. Go to "src" folder and add a new folder by name "slicers".

3. Add a new file "video-slicer.tsx"

4. Import "createSlice" and configure the slice with initial state and reducer actions.

5. Create Slice method comprises various properties
   a) actions
   b) reducer

   You have to export the actions and reducers

   video-slicer.tsx

```
import { createSlice } from "@reduxjs/toolkit";

const initialState = {
   videos : [],
   videosCount: 0
}

const videoSlice = createSlice({
   name: 'video',
   initialState,
   reducers: {
      addToSaveList : (state:any, action)=>{
         state.videos.push(action.payload);
         state.videosCount = state.videos.length;
      }
   }
});

export const {addToSaveList} = videoSlice.actions;
export default videoSlice.reducer;
```

6. Go to "src" and add a new folder by name "store"

7. Add a new file into store folder by name

"store.tsx"

8. Configure store by using redux toolkit "configureStore" function.
Store uses your video slicer and implements the reducers defined in slicer.

Note: The slicer is configured as default export, you have to import default and implement.

store.tsx

```
import { configureStore } from "@reduxjs/toolkit";
import videoSlicer from "../slicers/video-slicer";

export default configureStore({
   reducer: videoSlicer
});
```

9. Go to "main.tsx" and configure the provider by importing store from react-redux
[Provider locates value in memory and injects into component]

- Import provider & store [from your local store]

main.tsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import App from './App.tsx'
import '../node_modules/bootstrap/dist/css/bootstrap.css';
import '../node_modules/bootstrap-icons/font/bootstrap-icons.css';
import { CookiesProvider } from 'react-cookie';
import store from './store/store.tsx';
import { Provider } from 'react-redux';

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <CookiesProvider>
       <Provider store={store} >
         <App />
       </Provider>
    </CookiesProvider>
  </StrictMode>,
)
```

10. Go to your project user-dashboard component.

12. Import useDispatch(), which is responsible for dispatching the actions configured in reducer and update data into store.

13. Dispatch will carry the payload to store from component.

user-dash.tsx

```
import { useEffect, useState } from "react";
import { useCookies } from "react-cookie"
import { Link, useNavigate } from "react-router-dom";
import { VideoContract } from "../contracts/video-contract";
import axios from "axios";
import { addToSaveList } from "../slicers/video-slicer";
import { useDispatch} from "react-redux";

export function UserDash(){

  const [cookies, setCookie, removeCookie] = useCookies(['user_id']);
  const [videos, setVideos] = useState<VideoContract[]>();

  let navigate = useNavigate();
  const dispatch = useDispatch();

  useEffect(()=>{
    axios.get(`http://127.0.0.1:4040/videos`)
    .then(response=>{
       setVideos(response.data);
    });
  },[])

  function SignoutClick(){
    removeCookie('user_id');
    navigate('/');
  }

  function AddToWatchLaterClick(video:VideoContract){
    dispatch(addToSaveList(video));
  }


  return(
    <div>
      <h3 className="d-flex mt-4 justify-content-
between"><span>{cookies['user_id']}  <button className="bi bi-plus btn">My List</button>
</span> <span>User Dash</span> <button onClick={SignoutClick} className="btn btn-
link">  Signout</button> </h3>
      <div className="my-3 w-50">
        <div className="input-group">
          <input type="text" className="form-control" placeholder="Search videos: Java,
Aws, React" /> <button className="bi bi-search btn btn-warning"></button>
         </div>

      </div>
      <section className="d-flex flex-wrap">
        {
```

```
                    videos?.map(video=>
                        <div className="card m-2 p-2" style={{width:'300px'}} key={video.video_id}>
                            <div className="card-header">
                                <iframe width="100%" height="200" src={video.url}></iframe>
                            </div>
                            <div className="card-body">
                                <div className="fw-bold">{video.title}</div>
                                <p>{video.description}</p>
                            </div>
                            <div className="card-footer">
                                <button className="btn bi bi-hand-thumbs-up"> {video.likes} </button>
                                <button className="btn bi bi-hand-thumbs-down"> {video.dislikes}
</button>
                                <button className="btn bi bi-eye-fill"> {video.views} </button>
                                <button className="btn bi bi-plus " onClick={()=> {
AddToWatchLaterClick(video) } } > Watch Later</button>
                            </div>
                        </div>
                    )
                }
            </section>
        </div>
    )
}
```

Note : Download "redux-dev-tools" extension in your browser.

14. To access data from store you can import store in any component

```
    import store from "../store/store";

    store.getState().store.videos
    store.getState().store.videosCount
```

Implementing Callback & Memo in video library Project:
- Callback and Memo are used to save round trips.
- You can cache the data and use across multiple requests.
- Data is fetched from server only when there are changes identified on server.
- useMemo can store your data in memory.
- useCallback can store a function in memory.

Syntax:
```
   const ref = useMemo(()=>{
         // gets value
    },[dependency])
```

Session-60 (Testing and Deployment)

Testing & Deploying

- Testing is the process of verifying AS-IS & TO-BE.
- AS-IS refers to developer design & TO-BE is client requirement.

```
        AS-IS === TO-BE          => Test Pass
        AS-IS !== TO-BE          => Test Fail
```

- JavaScript based testing frameworks are required for testing React application
    - JEST
    - Jasmine Karma
    - VITest

- React application designed using JavaScript is enabled with JEST framework.

Testing a Component:
- Testing component comprises of 3 phases
    a) Arrange
    b) Act
    c) Assert

- Arrange is the process of configuring the component to test.
- Act defines the design and functionality to test.
- Assert is to verify test results and report the results.
- JEST framework provides various mock functions for arrange, act and assert

test()	It configures a case
render()	It renders the component to test
screen	It provides methods to access UI content
getBy..()	These are reference methods
toBe..()	These are assert methods

Ex:
weather.jsx
Weather Component
Client Requirements:
- Title must be "Weatherman"
- It must have a link to navigate to Google Weather
    <a> requires href with value "developers.google.com"

1. Add a new test file

```
        weather.test.js  [weather.spec.js]
```

2. Import the component and test functions

```
    import  { screen , render }  from  "@testing-library/react";
    import  { component   from "./component";
```

3. Configure the test case using "test()"

```
    test("title", ()=>{

    });
```

4. Render the component

```
    render(<Component/>);
```

5. Initialize the references for elements to test

```
    var  ref = screen.getBy..();
```

6. Configure the act

```
    expect(ref).toBe..();
    expect(ref).toHave..();
```

    weather.test.js

```
import { screen, render } from "@testing-library/react";
import { Weather } from "./weather";

// Title Test

test("Title Test",()=>{

    render(<Weather />);

    let title = screen.getByTestId("title");

    expect(title).toHaveTextContent(/Weatherman/);
});

// Link to Google test

test("Google Link Test", ()=>{

    render(<Weather />);

    let link = screen.getByText(/Google Weather/);

    expect(link).toBeInTheDocument();
    expect(link).toHaveAttribute("href", "https://developers.google.com&quot;);
});
```

7. Start testing

```
    > npm run test
```

Deploying
- It is the process of building application for production so that it can Go-Live.
- You can deploy on local server or cloud server.
- Local Servers
  a) IIS
  b) XAMP
  c) WAMP
  d) Tomcat etc..
- Cloud Servers
  a) Firebase
  b) AWS
  c) Azure
  d) Netlify
  e) GIT Hub Pages etc.

Ex: Deploying on Firebase [Google Cloud]

1. Login into your Firebase account with Google ID

   https://firebase.com

2. Go to "Console" and create a new project

    Name : weatherman-react-app

3. Install firebase tools on your PC

```
C:\>npm install  firebase-tools  -g
```

4. Go to your project terminal

   - Build your application for production

```
> npm run build
```

   - Login into firebase

```
> firebase login
```

   - Initialize firebase

```
> firebase init

?Firebase features do you want to set up :   Hosting
? Project : Use existing project
? Select Project : weatherman-react-app
? production folder : build
? re-write index.html : No
```

   - Deploy

```
> firebase deploy
```

Note: After making changes

```
> npm run build
> firebase deploy
```