**Repo**: <https://github.com/AjaySingala/ng-ts-node>

<https://youtu.be/CdE6rVfPJ9I>

Contents

[Node.js – Installation 10](#_Toc137641066)

[Windows 10](#_Toc137641067)

[Alternatives 10](#_Toc137641068)

[Node.js – Fundamentals 11](#_Toc137641069)

[Datatypes 11](#_Toc137641070)

[Loose Typing 11](#_Toc137641071)

[Objects & Functions 11](#_Toc137641072)

[Functions 12](#_Toc137641073)

[String and String Functions 13](#_Toc137641074)

[Buffer 13](#_Toc137641075)

[Node.js console-based application 13](#_Toc137641076)

[Node.js web-based application 14](#_Toc137641077)

[Node.js – NPM 16](#_Toc137641078)

[What is NPM? 16](#_Toc137641079)

[What is a Package? 16](#_Toc137641080)

[Download a Package 16](#_Toc137641081)

[Using a Package 16](#_Toc137641082)

[Node.js – package.json 17](#_Toc137641083)

[What is the purpose of package.json? 17](#_Toc137641084)

[Common fields in package.json 18](#_Toc137641085)

[Name 18](#_Toc137641086)

[version 18](#_Toc137641087)

[license 18](#_Toc137641088)

[author and contributors 18](#_Toc137641089)

[description 19](#_Toc137641090)

[keywords 19](#_Toc137641091)

[main 19](#_Toc137641092)

[scripts 19](#_Toc137641093)

[repository 19](#_Toc137641094)

[dependencies 20](#_Toc137641095)

[devDependencies 20](#_Toc137641096)

[Manage your package.json 20](#_Toc137641097)

[Typescript 22](#_Toc137641098)

[Features of TypeScript 22](#_Toc137641099)

[TypeScript and ECMAScript 23](#_Toc137641100)

[Why Use TypeScript? 23](#_Toc137641101)

[1) TypeScript improves your productivity while helping avoid bugs 23](#_Toc137641102)

[2) TypeScript brings the future JavaScript to today 24](#_Toc137641103)

[Components of TypeScript 24](#_Toc137641104)

[Install TS and Basic Compiling 25](#_Toc137641105)

[Using Proper NodeJS Installation 25](#_Toc137641106)

[Using npx ts-node 25](#_Toc137641107)

[Install VS Code 25](#_Toc137641108)

[Coding using TS 25](#_Toc137641109)

[First TS (using VS Code): 25](#_Toc137641110)

[With Live Server 26](#_Toc137641111)

[Transpiling Code from TS to JS with a Class: 27](#_Toc137641112)

[Sample with Interface 29](#_Toc137641113)

[1) Type system of TS 29](#_Toc137641114)

[2) Built-in types 30](#_Toc137641115)

[3) Interfaces 30](#_Toc137641116)

[4) Classes 31](#_Toc137641117)

[5) Inheritance 32](#_Toc137641118)

[6) Implements 33](#_Toc137641119)

[7) Static 34](#_Toc137641120)

[8) Abstract 35](#_Toc137641121)

[9) Modules 36](#_Toc137641122)

[10) enums 37](#_Toc137641123)

[11) Generics 38](#_Toc137641124)

[12) Namespaces 38](#_Toc137641125)

[13) Iterators 39](#_Toc137641126)

[Webpack 40](#_Toc137641127)

[SPA Definition 41](#_Toc137641128)

[What Is a Single Page Application? 41](#_Toc137641129)

[What Is Single Page Application Architecture? How Does It Work? 41](#_Toc137641130)

[Advantages of Single Page Applications 43](#_Toc137641131)

[Angular 45](#_Toc137641132)

[History 45](#_Toc137641133)

[What Is Angular? 45](#_Toc137641134)

[Why Do You Need a Framework? 45](#_Toc137641135)

[Features of Angular 45](#_Toc137641136)

[1. Document Object Model 45](#_Toc137641137)

[2. TypeScript 46](#_Toc137641138)

[3. Data Binding 46](#_Toc137641139)

[4. Testing 46](#_Toc137641140)

[Angular Architecture 46](#_Toc137641141)

[1. Modules 47](#_Toc137641142)

[2. Components 47](#_Toc137641143)

[3. Templates 47](#_Toc137641144)

[4. Metadata 47](#_Toc137641145)

[5. Services 47](#_Toc137641146)

[6. Dependency Injection 47](#_Toc137641147)

[AngularJS Directives 48](#_Toc137641148)

[Advantages of Angular 48](#_Toc137641149)

[1. Custom Components 48](#_Toc137641150)

[2. Data Binding 48](#_Toc137641151)

[3. Dependency Injection 48](#_Toc137641152)

[4. Testing 48](#_Toc137641153)

[5. Comprehensive 48](#_Toc137641154)

[6. Browser Compatibility 48](#_Toc137641155)

[Limitations of Angular 49](#_Toc137641156)

[1. Steep Learning Curve 49](#_Toc137641157)

[2. Limited SEO Options 49](#_Toc137641158)

[3. Migration 49](#_Toc137641159)

[4. Verbose and Complex 49](#_Toc137641160)

[Angular vs AngularJS 49](#_Toc137641161)

[Hello World – First Angular App 51](#_Toc137641162)

[Angular Prerequisites 51](#_Toc137641163)

[NodeJS 51](#_Toc137641164)

[Angular CLI 51](#_Toc137641165)

[Text Editor 51](#_Toc137641166)

[Creating the Angular HelloWorld Application 51](#_Toc137641167)

[Step 1 51](#_Toc137641168)

[**Step 2** 51](#_Toc137641169)

[Basics of an Angular App 52](#_Toc137641170)

[Root HTML - index.html 52](#_Toc137641171)

[The Entry Point - main.ts 52](#_Toc137641172)

[Main Module - app.module.ts 52](#_Toc137641173)

[Root Component – AppComponent – app.component.ts 52](#_Toc137641174)

[Angular Components 54](#_Toc137641175)

[What Are Angular Components? 54](#_Toc137641176)

[Creating Your First Angular Component 54](#_Toc137641177)

[Component Decorator Metadata 55](#_Toc137641178)

[Selector 56](#_Toc137641179)

[Template 56](#_Toc137641180)

[TemplateUrl 56](#_Toc137641181)

[Styles 56](#_Toc137641182)

[styleUrls 56](#_Toc137641183)

[Providers 56](#_Toc137641184)

[Animations 56](#_Toc137641185)

[Demo: Creating an Angular Component 56](#_Toc137641186)

[Angular – Data Binding 58](#_Toc137641187)

[What Is Angular Data Binding? 58](#_Toc137641188)

[AngularJS Controller 58](#_Toc137641189)

[Attribute Binding 58](#_Toc137641190)

[Class Binding 58](#_Toc137641191)

[Style Binding 58](#_Toc137641192)

[ngModel 58](#_Toc137641193)

[Types of Data Binding 58](#_Toc137641194)

[Interpolation Binding 58](#_Toc137641195)

[Property Binding 59](#_Toc137641196)

[Event Binding 59](#_Toc137641197)

[Two-way Data Binding 60](#_Toc137641198)

[Angular – Components – Input and Output 62](#_Toc137641199)

[Angular @Input 62](#_Toc137641200)

[Angular @Output 62](#_Toc137641201)

[**~~Step 1~~**~~: Create a new project or use an existing one.~~ 62](#_Toc137641202)

[~~Step 2: Create parent and child components.~~ 62](#_Toc137641203)

[**~~Step 3~~**~~: Define HTML for parent component.~~ 63](#_Toc137641204)

[~~Step 4: Define HTML for child components.~~ 63](#_Toc137641205)

[~~Step 5: Use Input to display parent component value~~ 64](#_Toc137641206)

[~~Step 6: Pass value from child to parent component.~~ 65](#_Toc137641207)

[~~Full Code~~ 66](#_Toc137641208)

[Another Example 67](#_Toc137641209)

[Create components 67](#_Toc137641210)

[main.component.ts 67](#_Toc137641211)

[main.component.html 67](#_Toc137641212)

[sub.component.ts 67](#_Toc137641213)

[sub.component.html 68](#_Toc137641214)

[Explanation 68](#_Toc137641215)

[Angular – Directives 70](#_Toc137641216)

[Attribute directives 70](#_Toc137641217)

[Structural directives 70](#_Toc137641218)

[Component based directives 70](#_Toc137641219)

[DOM Overview 70](#_Toc137641220)

[Structural directives 71](#_Toc137641221)

[NgIf directive 71](#_Toc137641222)

[ngIfElse directive 71](#_Toc137641223)

[ngFor directive 72](#_Toc137641224)

[ngFor odd, even, first, last 73](#_Toc137641225)

[~~trackBy~~ 77](#_Toc137641226)

[NgSwitch directive 78](#_Toc137641227)

[Attribute directives 79](#_Toc137641228)

[ngStyle 79](#_Toc137641229)

[ngClass 80](#_Toc137641230)

[Custom directives 81](#_Toc137641231)

[ng-template 82](#_Toc137641232)

[ng-template with structural directive 82](#_Toc137641233)

[NgForOf directive 82](#_Toc137641234)

[ngFor and ngForOf Example 83](#_Toc137641235)

[Creating a Custom Directive 85](#_Toc137641236)

[Directive Decorator 85](#_Toc137641237)

[Attribute Selector 86](#_Toc137641238)

[Directive Constructor 86](#_Toc137641239)

[Full Code 87](#_Toc137641240)

[Component directives 89](#_Toc137641241)

[Sample Working example 89](#_Toc137641242)

[Bind Select Dropdown List 93](#_Toc137641243)

[Angular – Dependency Injection (DI) and Services 95](#_Toc137641244)

[Motivation 95](#_Toc137641245)

[Injectors 95](#_Toc137641246)

[Dependency Injection 95](#_Toc137641247)

[In Any Case 96](#_Toc137641248)

[Service 97](#_Toc137641249)

[Module, Directive, and Component 97](#_Toc137641250)

[Instantiating References 97](#_Toc137641251)

[Instantiating Services 98](#_Toc137641252)

[Promises in TypeScript 101](#_Toc137641253)

[Promises 101](#_Toc137641254)

[Promise Chaining 103](#_Toc137641255)

[Promise.resolve 104](#_Toc137641256)

[Promise.reject 104](#_Toc137641257)

[Promise.all 105](#_Toc137641258)

[Promise.allSettled 106](#_Toc137641259)

[Promise.race 109](#_Toc137641260)

[Async/Await keywords 110](#_Toc137641261)

[Angular – HttpClient 115](#_Toc137641262)

[Why Do We Need HttpClient? 115](#_Toc137641263)

[What Is HttpClient? 115](#_Toc137641264)

[Features of HttpClient 115](#_Toc137641265)

[What Is an RxJS Observable? 115](#_Toc137641266)

[How To Use HttpClient in Angular? 115](#_Toc137641267)

[Another Example 119](#_Toc137641268)

[Service Setup 119](#_Toc137641269)

[Making the API call 119](#_Toc137641270)

[User Interface 121](#_Toc137641271)

[Updated Component 121](#_Toc137641272)

[Result 124](#_Toc137641273)

[Angular – HttpClient – CRUD 126](#_Toc137641274)

[Angular – HttpClient – Calling .NET Core API (with CORS) 130](#_Toc137641275)

[Angular – Pipes 131](#_Toc137641276)

[Create the app and component 131](#_Toc137641277)

[Adding Parameters 131](#_Toc137641278)

[Add Date pipe 131](#_Toc137641279)

[Parameters in Date 131](#_Toc137641280)

[Chained pipes 131](#_Toc137641281)

[Built-in Pipes 132](#_Toc137641282)

[AsyncPipe 132](#_Toc137641283)

[CurrencyPipe 132](#_Toc137641284)

[SlicePipe 133](#_Toc137641285)

[DecimalPipe 133](#_Toc137641286)

[Formatting values 134](#_Toc137641287)

[PercentPipe 136](#_Toc137641288)

[JsonPipe 136](#_Toc137641289)

[Creating custom pipe 137](#_Toc137641290)

[Angular – Routing 138](#_Toc137641291)

[Introducing Angular Router 138](#_Toc137641292)

[THE ROUTER-OUTLET 138](#_Toc137641293)

[ROUTES AND PATHS 138](#_Toc137641294)

[ROUTE MATCHING STRATEGIES 138](#_Toc137641295)

[ROUTE PARAMS 139](#_Toc137641296)

[ROUTE GUARDS 139](#_Toc137641297)

[NAVIGATION DIRECTIVE 139](#_Toc137641298)

[MULTIPLE OUTLETS AND AUXILIARY ROUTES 139](#_Toc137641299)

[Creating an Angular Project with Routing 140](#_Toc137641300)

[Creating A Fake Back-End Service 140](#_Toc137641301)

[Creating Our Angular Components 142](#_Toc137641302)

[Setting Up Routing 142](#_Toc137641303)

[IMPORTING THE ROUTING MODULE 143](#_Toc137641304)

[CREATING ROUTES 143](#_Toc137641305)

[ADDING NAVIGATION LINKS 144](#_Toc137641306)

[Angular – Forms 147](#_Toc137641307)

[Template driven forms 147](#_Toc137641308)

[Create simple form 147](#_Toc137641309)

[Configure Forms 147](#_Toc137641310)

[Reactive Forms 148](#_Toc137641311)

[Configure Reactive forms 148](#_Toc137641312)

[Create Reactive forms 148](#_Toc137641313)

[Forms Validation 149](#_Toc137641314)

[Angular – Deployment 151](#_Toc137641315)

[Minimal Todo Angular App Deployment to Azure using GitHub Actions and SonarCloud 151](#_Toc137641316)

[Local Sonar 151](#_Toc137641317)

[Using GitHub Actions 152](#_Toc137641318)

[Adding Environments and Approval Steps 153](#_Toc137641319)

[Angular – Unit Testing 155](#_Toc137641320)

[What is Angular Unit testing? 155](#_Toc137641321)

[Unit Testing with Jasmine and Karma for Angular Apps 155](#_Toc137641322)

[How to create an Angular test App? 155](#_Toc137641323)

[How to write a unit test in Angular? 155](#_Toc137641324)

[How to write a negative unit test in Angular? 157](#_Toc137641325)

[Angular – Unit Testing – TestBed 159](#_Toc137641326)

[Fixtures and DI 159](#_Toc137641327)

[Test Specs 160](#_Toc137641328)

[When to Use ATB 160](#_Toc137641329)

[Component class testing 160](#_Toc137641330)

[Component and Associated Class Testing 162](#_Toc137641331)

[Testing Components with Dependencies (mocking a service) 164](#_Toc137641332)

[Mocking 166](#_Toc137641333)

[Testing with the Real AuthService 167](#_Toc137641334)

[Mocking with Fake Classes 168](#_Toc137641335)

[Mocking by Overriding Functions 170](#_Toc137641336)

[Mocking with Spies 170](#_Toc137641337)

[Angular – Unit Testing – Code Coverage 172](#_Toc137641338)

# Node.js – Installation

<https://nodejs.org/en/#home-downloadhead>

<https://nodejs.org/en/download/package-manager/>

## Windows

Download the [Windows Installer](https://nodejs.org/en/#home-downloadhead) directly from the [nodejs.org](https://nodejs.org/) web site.

### Alternatives

Using [**Winget**](https://aka.ms/winget-cli):

winget install OpenJS.NodeJS

# or for LTS

winget install OpenJS.NodeJS.LTS

After running one of the two commands above, it may be necessary to restart the terminal emulator before the node CLI command becomes available.

Using [**Chocolatey**](https://chocolatey.org/):

cinst nodejs

# or for full install with npm

cinst nodejs.install

Using [**Scoop**](https://scoop.sh/):

scoop install nodejs

# or for LTS

scoop install nodejs-lts

# Node.js – Fundamentals

Node.js is a cross-platform JavaScript runtime environment. It allows the creation of scalable Web servers without threading and networking tools using JavaScript and a collection of “modules” that handle various core functionalities. It can make console-based and web-based node.js applications.

## Datatypes

Node.js contains various types of data types similar to JavaScript.

* Boolean
* Undefined
* Null
* String
* Number

## Loose Typing

Node.js supports loose typing, it means you don’t need to specify what type of information will be stored in a variable in advance. We use *var* keyword in Node.js to declare any type of variable. Examples are given below:

**All code in /Angular/Node.JS Demos**

**Example: loose\_typing.js**

|  |
| --- |
| // Variable store number data type  **var** a = 35;  console.log(**typeof** a);    // Variable store string data type  a = "GeeksforGeeks";  console.log(**typeof** a);    // Variable store Boolean data type  a = **true**;  console.log(**typeof** a);    // Variable store undefined (no value) data type  a = undefined;  console.log(**typeof** a); |

**Run as:**

**node loose\_typing.js**

**Output:**

number

string

boolean

undefined

## **Objects & Functions**

Node.js objects are same as JavaScript objects i.e. the objects are similar to variable and it contains many values which are written as **name : value** pairs. Name and value are separated by colon and every pair is separated by comma.

**Example: objects\_and\_functions.js**

|  |
| --- |
| **var** company = {      Name: "ajay singala",      Address: "Noida",      Contact: "+919876543210",      Email: "abc@ajay.org"  };    // Display the object information  console.log("Information of variable company:", company);    // Display the type of variable  console.log("Type of variable company:", **typeof** company); |

**Output:**

Information of variable company: {

Name: ajay singala,

Address: 'Noida',

Contact: '+919876543210',

Email: 'abc@ajay.org'

}

Type of variable company: object

## **Functions**

Node.js functions are defined using **function** keyword then the name of the function and parameters which are passed in the function. In Node.js, we don’t have to specify datatypes for the parameters and check the number of arguments received. Node.js functions follow every rule which is there while writing JavaScript functions.

**Example: multiply.js**

|  |
| --- |
| **function** multiply(num1, num2) {        // It returns the multiplication      // of num1 and num2  **return** num1 \* num2;  }    // Declare variable  **var** x = 2;  **var** y = 3;    // Display the answer returned by  // multiply function  console.log("Multiplication of", x,      "and", y, "is", multiply(x, y)); |

**Output:**

Multiplication of 2 and 3 is 6

If you observe in the above example, we have created a function called “multiply” with parameters same as JavaScript.

## **String and String Functions**

In Node.js we can make a variable as string by assigning a value either by using single (”) or double (“”) quotes and it contains many functions to manipulate to strings.  
Following is the example of defining string variables and functions in node.js.

**Example: string\_funcs.js**

|  |
| --- |
| **var** x = "Welcome to ajay singala ";  **var** y = 'Node.js Tutorials';  **var** z = [this, 'is, node'];  console.log(x);  console.log(y);  console.log("Concat Using (+) :", (x + y));  console.log("Concat Using Function :", (x.concat(y)));  console.log("Split string: ", x.split(' '));  console.log("Join string: ", z.join(', '));  console.log("Char At Index 5: ", x.charAt(5) ); |

**Output:**

Welcome to ajay singala

Node.js Tutorials

Concat Using (+) : Welcome to ajay singala Node.js Tutorials

Concat Using Function : Welcome to ajay singala Node.js Tutorials

Split string: [ 'Welcome', 'to', 'ajay', 'singala', '' ]

Join string: [object Object], is, node

Char At Index 5: m

## **Buffer**

In node.js, we have a data type called “Buffer” to store a binary data and it is useful when we are reading a data from files or receiving a packets over network.

## **Node.js console-based application**

Make a file called console.js with the following code.

|  |
| --- |
| console.log('Hello this is the console-based application');  console.log('This all will be printed in console');  // The above two lines will be printed in the console. |

To run this file, open node.js command prompt and go to the folder where console.js file exist and write the following command. It will display content on console.

Text

Description automatically generated

The console.log() method of console class prints the message passed in the method in the console.

## **Node.js web-based application**

Node.js web application contains different types of modules which is imported using **require()** directive and we have to create a server and write code for the read request and return response.

Make a file node\_webapp.js with the following code.

|  |
| --- |
| // Require http module  **var** http = require("http");    // Create server  http.createServer(**function** (req, res) {        // Send the HTTP header      // HTTP Status: 200 : OK      // Content Type: text/plain      res.writeHead(200, {'Content-Type': 'text/plain'});        // Send the response body as "This is the example      // of node.js web based application"     res.end('This is the example of node.js web-based application \n');    // Console will display the message  }).listen(5000,      ()=>console.log('Server running at <http://127.0.0.1:5000/>')); |

To run this file, follow the steps as given below:

* Search the node.js command prompt in the search bar and open the node.js command prompt.
* Go to the folder using **cd** command in command prompt and write the following command **node web.js**  
  Text

  Description automatically generated
* Now the server has started and go to the browser and open this url **localhost:5000**  
  Graphical user interface, text, application

  Description automatically generated

You will see the response which you have sent back from web.js in the browser. If any changes are made in the node\_webapp.js file then again run the command **node node\_webapp.js** and refresh the tab in the browser.

# Node.js – NPM

## What is NPM?

NPM is a package manager for Node.js packages, or modules if you like. [www.npmjs.com](https://www.npmjs.com/) hosts thousands of free packages to download and use. The NPM program is installed on your computer when you install Node.js

## What is a Package?

A package in Node.js contains all the files you need for a module.

Modules are JavaScript libraries you can include in your project.

## Download a Package

Downloading a package is very easy.

Open the command line interface and tell NPM to download the package you want.

I want to download a package called "upper-case":

**Download "upper-case":**

C:\Users\Your Name>npm install upper-case

Now you have downloaded and installed your first package!

NPM creates a folder named "node\_modules", where the package will be placed. All packages you install in the future will be placed in this folder.

My project now has a folder structure like this:

C:\Users\My Name\node\_modules\upper-case

## Using a Package

Once the package is installed, it is ready to use.

Include the "upper-case" package the same way you include any other module:

var uc = require('upper-case');

Create a Node.js file that will convert the output "Hello World!" into upper-case letters:

**Example**

var http = require('http');

var uc = require('upper-case');

http.createServer(function (req, res) {

  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write(**uc.upperCase("Hello World!")**);

  res.end();

}).listen(8080);

Save the code above in a file called "demo\_uppercase.js", and initiate the file:

**Initiate demo\_uppercase:**

C:\Users\Your Name>node demo\_uppercase.js

If you have followed the same steps on your computer, you will see the same result as the example: [http://localhost:8080](http://localhost:8080/)

# Node.js – package.json

<https://heynode.com/tutorial/what-packagejson/>

If you've worked with Node.js before, you have likely encountered a package.json file. It is a [JSON](http://www.json.org/) file that lives in the root directory of your project. Your package.json holds important information about the project. It contains human-readable metadata about the project (like the project name and description) as well as functional metadata like the package version number and a list of dependencies required by the application.

An example package.json might look like this:

{

"name": "my-project",

"version": "1.5.0",

"description": "Express server project using compression",

"main": "src/index.js",

"scripts": {

"start": "node index.js",

"dev": "nodemon",

"lint": "eslint \*\*/\*.js"

},

"dependencies": {

"express": "^4.16.4",

"compression": "~1.7.4"

},

"devDependencies": {

"eslint": "^5.16.0",

"nodemon": "^1.18.11"

},

"repository": {

"type": "git",

"url": "https://github.com/osiolabs/example.git"

},

"author": "Jon Church",

"contributors": [{

"name": "Amber Matz",

"email": "example@example.com",

"url": "https://www.osiolabs.com/#team"

}],

"keywords": ["server", "osiolabs", "express", "compression"]

}

## What is the purpose of package.json?

Your project's package.json is the central place to configure and describe how to interact with and run your application. It is used by the npm CLI (and yarn) to identify your project and understand how to handle the project's dependencies. It's the package.json file that enables npm to start your project, run scripts, install dependencies, publish to the NPM registry, and many other useful tasks. The npm CLI is also the best way to manage your package.json because it helps generate and update your package.json file throughout a project's life.

Your package.json fills several roles in the lifecycle of your project, some of which only apply for packages published to NPM. If you're not publishing your project to the NPM registry or otherwise making it publicly available to others, your package.json is still essential to the development flow.

Your project also must include a package.json before any packages can be installed from NPM. This is probably the top reason why you need one in your project.

## Common fields in package.json

Let's look at some of the most common and important fields that can be in a package.json, to better understand how to use and manage this essential file. Some are required for publishing to NPM, while others help the npm CLI run the application or install dependencies.

There are more fields than the ones we cover, and you can read about the rest in the [documentation](https://docs.npmjs.com/files/package.json), but these are the essential package.json properties to understand.

### Name

"name": "my-project"

The name field defines the name of the package. When publishing to the NPM registry, this is the name the package will be listed under. It must be no more than 214 characters, only lowercase letters, and it must be URL-safe (hyphens and underscores allowed, but no spaces or other characters disallowed in URLs).

If publishing your package to NPM, the name property is required and must be unique. You'll receive an error if trying to publish a package under a name that is currently used on the NPM registry. If you aren't developing a package which you'll eventually publish, the name does not have to be unique.

### version

"version": "1.5.0",

The version field is very important for any published package, and required before publishing. It is the current version of the software that the package.json is describing.

You are not required to use [SemVer](https://docs.npmjs.com/about-semantic-versioning), but it is the standard used in the Node.js ecosystem and highly recommended. For an unpublished package, this property isn't strictly required. Typically, the version number is bumped according to SemVer before publishing new versions to NPM. This workflow isn't typically used when a package is not being relied upon as a dependency, or the package isn't being published to NPM. But if a package is being used as a dependency, keeping the version field up to date is very important to make sure others are using the proper version of a package.

### license

This is a very important but often overlooked property. The license field lets us define what license applies to the code the package.json is describing. Again, this is very important when publishing a project to the NPM registry, as the license may limit the use of your software by some developers or organizations. Having a clear license in place helps clearly define what terms the software is able to be used under.

The value of this field will usually be the license's identifier code -- a string like "MIT" or "ISC" for the [MIT](https://opensource.org/licenses/MIT) license and [ISC](https://opensource.org/licenses/ISC) license respectively. If you don't wish to provide a license, or explicitly do not want to grant use of a private or unpublished package, you can put "UNLICENSED" as the license. [Choose a License](https://choosealicense.com/) is a helpful resource if you're not sure which license to use.

### author and contributors

"author": "Jon Church jon@example.com https://www.osioslabs.com/#team",

"contributors": [{

"name": "Amber Matz",

"email": "example@example.com",

"url": "https://www.osiolabs.com/#team"

}],

The author and contributor fields function similarly. They are both "people" fields which can be either a string in the format of "Name <email> <url>" , or an object with fields name, email, url. The email and url are both optional.

Author is for a single person, and contributors is an array of people.

These fields are a useful way to list contacts for a public project, as well as share credit with contributors.

### description

The description field is used by the NPM registry for published packages, to describe the package in search results and on the [npmjs.com](https://www.npmjs.com/) website.

This string is used to help surface packages when users search the NPM registry. This should be a short summary of what the package is for.

It can also be useful as simple documentation for your project, even if you aren't publishing it to the NPM registry.

### keywords

"keywords": ["server", "osiolabs", "express", "compression"]

The keywords field is an array of strings, and serves a similar purpose to the description. This field is indexed by the NPM registry to help find packages when someone searches for them. Each value in the array is one keyword associated with your package.

This field doesn't have much use if you're not publishing to the NPM registry, and you can feel free to omit it.

### main

"main": "src/index.js",

The main field is a functional property of your package.json. This defines the entry point into your project, and commonly the file used to start the project.

If your package (let's say its name is foo-lib) is installed by a user, then when a user does require('foo-lib'), it is the module.exports property of the file listed in the main field that is returned by require.

This is commonly an index.js file in the root of your project, but it can be any file you choose to use as the main entry-point to your package.

### scripts

"scripts": {

"start": "node index.js",

"dev": "nodemon"

}

The scripts field is another functional piece of metadata in your package.json. The scripts property takes an object with its keys being scripts we can run with npm run <scriptName>, and the value is the actual command which is run. These are typically terminal commands, which we put into the scripts field so we can both document them and reuse them easily.

Scripts are powerful tools that the npm CLI can use to run tasks for your project. They can do the job of most task runners used during development.

### repository

"repository": {

"type": "git",

"url": "https://github.com/osiolabs/example.git"

}

You can record the repository the code for a project lives in by providing the repository field. This field is an object which defines the url where the source code is located, and what type of version control system it uses. For open source projects, this is likely GitHub or Bitbucket with Git as the version control system.

An important note is that the URL field is meant to point to where the version control can be accessed from, not just the released code base.

### dependencies

"dependencies": {

"express": "^4.16.4",

"compression": "~1.7.4"

}

This is one of the most important fields in your package.json, and likely the entire reason you need one. All of the dependencies your project uses (the external code that the project relies on) are listed here. When a package is installed using the npm CLI, it is downloaded to your node\_modules/ folder and an entry is added to your dependencies property, noting the name of the package and the installed version.

The dependency field is an object with package names as keys, and a version or version range as a value. From this list, npm knows what packages to fetch and install (and at what versions) when npm install is run in the directory. The dependency field of your package.json is at the heart of your project, and defines the external packages your project requires.

The carets (^) and tildes (~) you see in the dependency versions are notation for version ranges defined in SemVer.

### devDependencies

"devDependencies": {

"nodemon": "^1.18.11"

}

Similar to the dependencies field, but for packages which are only needed during development, and aren't needed in production.

An example would be using a tool to reload your project during development, like [nodemon](https://www.npmjs.com/package/nodemon), which we have no use for once the application is deployed and in production. The devDependencies property lets us explicitly note which dependencies aren't needed in production. When installing your app in a production environment, you can use npm install --production to only install what is listed in the dependency field of package.json.

Recording a devDependency is a great way to document what tools are needed for the app during development. To install a package from npm as a devDependency, you can run npm install --save-dev <package>.

There is another way the devDependencies property is useful to us, and that's by using them in our npm scripts.

## Manage your package.json

A package.json file must be valid JSON. This means any missing commas, unclosed quotes, or other formatting errors will prevent npm from interacting with the package.json. If you do introduce an error, the next time you run an npm command you will see an error from npm. It's recommended to use the npm CLI for updating and managing your package.json when possible, to avoid accidentally introducing errors to your package.json, and to make managing your dependencies easier.

Using npm init to [create your package.json](https://heynode.com/tutorial/create-packagejson-file) will help to ensure you generate a valid file.

Dependencies are best managed by using npm's commands npm install, npm uninstall, and npm update, so your package.json and node\_modules/ folder are kept in sync. Manually adding a dependency listing will require you to run npm install before the dependency is actually installed to your project.

Because our package.json is only where we record dependencies, and our node\_modules/ folder is where the actual code for dependencies is installed, manually updating the dependency field of package.json does not immediately reflect the state of our node\_modules/ folder. That's why you want to use npm to help manage dependencies, because it will update both the package.json and node\_modules/ folder in tandem.

You can always edit your package.json manually in your text editor and make changes. That works well for most fields, so long as you're careful not to introduce any JSON formatting errors. We recommend you use the npm CLI commands wherever you can, however.

# Typescript

**All code in /Angular/TypeScript Demos**

TypeScript is Typed JavaScript. TypeScript adds types to JavaScript to help you speed up the development by catching errors before you even run the JavaScript code.

TypeScript is an open-source programming language that builds on top of JavaScript. It works on any browser, any OS, any environment that JavaScript runs.

* TS is a pre-compiler (Developed by Microsoft, open source).
* Browser does not recognize TS
* It compiles in to JavaScript that can then be used (in .NET Core etc.)
* The new Angular versions are very different from Angular1 (AngularJS)

TypeScript is a super set of JavaScript.

TypeScript builds on top of JavaScript. First, you write the TypeScript code. Then, you compile the TypeScript code into plain JavaScript code using a TypeScript compiler.

Once you have the plain JavaScript code, you can deploy it to any environments that JavaScript runs.

TypeScript files use the .ts extension rather than the .js extension of JavaScript files.

A picture containing chart

Description automatically generated

TypeScript uses the JavaScript syntaxes and adds additional syntaxes for supporting Types.

If you have a JavaScript program that doesn’t have any syntax errors, it is also a TypeScript program. It means that all JavaScript programs are TypeScript programs. This is very helpful if you’re migrating an existing JavaScript codebase to TypeScript.

The following diagram shows the relationship between TypeScript and JavaScript:

Diagram

Description automatically generated

## Features of TypeScript

**TypeScript is just JavaScript**. TypeScript starts with JavaScript and ends with JavaScript. Typescript adopts the basic building blocks of your program from JavaScript. Hence, you only need to know JavaScript to use TypeScript. All TypeScript code is converted into its JavaScript equivalent for the purpose of execution.

**TypeScript supports other JS libraries**. Compiled TypeScript can be consumed from any JavaScript code. TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.

**JavaScript is TypeScript**. This means that any valid **.js** file can be renamed to **.ts** and compiled with other TypeScript files.

**TypeScript is portable**. TypeScript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on. Unlike its counterparts, TypeScript doesn’t need a dedicated VM or a specific runtime environment to execute.

## TypeScript and ECMAScript

The ECMAScript (ES) (**European Computer Manufacturers Association** (**ECMA**)) specification is a standardized specification of a scripting language. There are six editions of ECMA-262 published. Version 6 of the standard is codenamed "Harmony". TypeScript is aligned with the ECMAScript6 specification.



The ECMAScript specification is a standardized specification of a scripting language developed by [Brendan Eich](https://en.wikipedia.org/wiki/Brendan_Eich) of [Netscape](https://en.wikipedia.org/wiki/Netscape_Communications_Corporation); initially it was named Mocha, later LiveScript, and finally JavaScript. In December 1995, [Sun Microsystems](https://en.wikipedia.org/wiki/Sun_Microsystems) and Netscape announced JavaScript in a press release.

## Why Use TypeScript?

The main goals of TypeScript are:

* Introduce optional types to JavaScript.
* Implement planned features of future JavaScript, a.k.a. ECMAScript Next or ES Next to the current JavaScript.

### 1) TypeScript improves your productivity while helping avoid bugs

Types increase productivity by helping you avoid many mistakes. By using types, you can catch bugs at the compile-time instead of having them occurring at runtime.

The following function adds two numbers x and y:

function add(x, y) {

return x + y;

}

Code language: JavaScript (javascript)

If you get the values from HTML input elements and pass them into the function, you may get an unexpected result:

let result = add(input1.value, input2.value);

console.log(result); // result of concatenating strings

For example, if users entered 10 and 20, the add() function would return 1020, instead of 30.

The reason is that the input1.value and input2.value are strings, not numbers. When you use the operator + to add two strings, it concatenates them into a single string.

When you use TypeScript to explicitly specify the type for the parameters like this:

function add(x: number, y: number) {

return x + y;

}

In this function, we added the number types to the parameters. The function add() will accept only numbers, not any other values.

When you invoke the function as follows:

let result = add(input1.value, input2.value);

… the TypeScript compiler will issue an error if you compile the TypeScript code into JavaScript. Hence, you can prevent the error from happening at runtime.

### 2) TypeScript brings the future JavaScript to today

TypeScript supports the upcoming features planned in the ES Next for the current JavaScript engines. It means that you can use the new JavaScript features before web browsers (or other environments) fully support them.

Every year, TC39 releases several new features for ECMAScript, which is the standard of JavaScript. The feature proposals typically go through five stages:

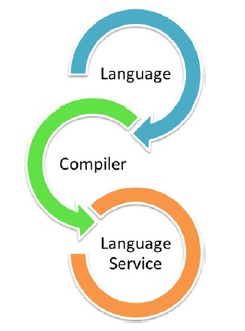
* Stage 0: Strawperson
* Stage 1: Proposal
* Stage 2: Draft
* Stage 3: Candidate
* Stage 4: Finished

And TypeScript generally supports features that are in the stage 3.

## Components of TypeScript

At its heart, TypeScript has the following three components −

* **Language** − It comprises of the syntax, keywords, and type annotations.
* **The TypeScript Compiler** − The TypeScript compiler (tsc) converts the instructions written in TypeScript to its JavaScript equivalent.
* **The TypeScript Language Service** − The "Language Service" exposes an additional layer around the core compiler pipeline that are editor-like applications. The language service supports the common set of a typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.



[www.typescriptlang.org/handbook](http://www.typescriptlang.org/handbook)

## Install TS and Basic Compiling

### Using Proper NodeJS Installation

* Requires Node.js
* Check if NodeJS + NodeJS Package Manager (npm) is installed or not

node -v

npm -v

Or install from <https://nodejs.org/en/>

* Install TS:

npm install -g typescript

* + OR go to typescriptlang.org and install the editor
  + -g means install globally, so available to all projects.
* run TSC to check TS installation

tsc -version

OR

tsc --v

* Transpiler / Transpilation
  + Process of compiling Typescript (.ts) to Javascript (.js)
* When transpiling TS programs from Powershell or VS Code, you will get an error like this:
  + To get rid of this error, open Powershell in Admin mode and run:

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned

### Using npx ts-node

* Requires Node.js
* Create a .ts file
* Run it with:

npx ts-node <filename.ts>

### Install VS Code

To install the VS Code, you follow these steps:

* Navigate to the [VS Code download page](https://code.visualstudio.com/download).
* Download the latest version of VS Code that suits your OS (Windows, macOS, or Linux)
* Execute the downloaded package or the installer file to launch the setup wizard. The installation process is also quite straightforward.
* Launch the VS Code.

To install the **Live Server** extension, you follow these steps:

* Click the **Extensions**tab to find the extensions for VS Code.
* Type the **live server** to search for it.
* Click the **install**button to install the extension.

## Coding using TS

### First TS (using VS Code):

Create a folder "helloWorld" and navigate to that

Create file called helloWorld.ts

console.log("Hello World");

from cmd prompt, TSC helloWorld.ts

creates helloWorld.js

run with node helloWorld.js

Add a func:

// remove the console.log("Hello World");

function hello(string: String) {

console.log("Hello " + string);

}

hello("Ajay");

from cmd: node helloWorld.js

shows Hello World

but node helloWorld.ts will not work as node does not recognize TS, only JS

so, TSC helloWorld.ts, updates the js file

run node helloWorld.js

shows Hello Ajay

change code to: hello(12)

tsc helloWorld.ts

error: number not assignable to param of type string

change to hello("Neo");

compile and run. works.

### With Live Server

<https://www.typescripttutorial.net/typescript-tutorial/typescript-hello-world/>

* First, create a new folder to store the code, e.g., helloworld.
* Second, launch VS Code and open that folder.
* Third, create a new TypeScript file called app.ts. The extension of a TypeScript file is .ts.
* Fourth, type the following source code in the app.ts file:

let message: string = 'Hello, World!';

console.log(message);

* Fifth, launch a new Terminal within the VS Code by using the keyboard shortcut Ctrl+` or follow the menu Terminal > New Terminal
* Sixth, type the following command on the Terminal to compile the app.ts file:

tsc app.ts

If everything is fine, you’ll see a new file called app.js is generated by the TypeScript compiler.

To run the app.js file in node.js, you use the following command:

node app.js

If you installed the ts-node module mentioned in the [setting up TypeScript development environment](https://www.typescripttutorial.net/typescript-tutorial/setup-typescript/), you can use just one command to compile the TypeScript file and execute the output file in one shot:

ts-node app.ts

#### TypeScript Hello World program in Web Browsers

You follow these steps to create a webpage that shows the Hello, World! message on web browsers.

* First, create a new file called index.html and include the app.js as follows:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>TypeScript: Hello, World!</title>

</head>

<body>

<script src="app.js"></script>

</body>

</html>

* Second, change the app.ts code to the following:

let message: string = 'Hello, World!';

// create a new heading 1 element

let heading = document.createElement('h1');

heading.textContent = message;

// add the heading the document

document.body.appendChild(heading);

* Third, compile the app.ts file:

tsc app.ts

* Fourth, open the Live Server from the VS code by right-mouse click the index.html and select the Open with Live Server option:

The Live Server will open the index.html with the following message:

Graphical user interface, text, application, chat or text message

Description automatically generated

To make the changes, you need to edit the app.ts file. For example:

let message: string = 'Hello, TypeScript!';

let heading = document.createElement('h1');

heading.textContent = message;

document.body.appendChild(heading);

And compile the app.ts file:

tsc app.ts

The TypeScript compiler will generate a new app.js file, and the Live Server will automatically reload it on the web browser.

Note that the app.js is the output file of the app.ts file, therefore, you should never directly change the code in this file, or you’ll lose the changes once you recompile the app.ts file.

### Transpiling Code from TS to JS with a Class:

* Create a file person.ts with the following code:

class Person {

public firstName: string;

public lastName: string;

constructor (firstName: string, lastName: string) {

this.firstName = firstName;

this.lastName = lastName;

}

}

* Compile with tsc person.ts
* Generates person.js
* Open it and show
* Add a private field and a function accessor to person.ts

class Person {

public firstName: string;

public lastName: string;

private \_fullName: string;

constructor (firstName: string, lastName: string) {

this.firstName = firstName;

this.lastName = lastName;

}

get fullName(): string {

return `${this.firstName} ${this.lastName}`;

}

}

* Compile with tsc person.ts
  + Gives an error
  + The proper way to define an accessor in javascript is using   Object.defineProperty() that is only available in ES5 and onward, but as default, the transpiler tries to create ES3 code. To overcome this problem we have to tell the transpiler that it should target ES5 and not ES3.
* Compile with tsc person.ts --target ES5
  + Works! Show the person.js file
* Change the code to the following:

class Person {

public firstName: string;

public lastName: string;

private \_fullName: string;

constructor(firstName: string, lastName: string) {

this.firstName = firstName;

this.lastName = lastName;

}

fullName(): string {

return `${this.firstName} ${this.lastName}`;

}

}

var obj = new Person('John', "Smith");

console.log(obj.fullName());

* Compile with tsc person.ts
* Run with node person.js
  + Works!!!

### Sample with Interface

A) greeter.ts:

class Student {

fullName: string;

constructor(public firstName: string, public middleInitial: string, public lastName: string) {

this.fullName = firstName + " " + middleInitial + " " + lastName;

}

}

interface Person {

firstName: string;

lastName: string;

}

function greeter(person : Person) {

return "Hello, " + person.firstName + " " + person.lastName;

}

let user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);

* Run TSC greeter.ts
* Generates greeter.js

B) greeter.html

<!DOCTYPE html>

<html>

<head><title>TypeScript Greeter</title></head>

<body>

<script src="greeter.js"></script>

</body>

</html>

C) Open greeter.html in browser.

### Type system of TS

Basic types intro

new folder, types

file types.ts

define type with a colon

var n: Number = 1;

compile and show .js

does not have "Number"

n = "Ajay"

compile, gives error

but .js has n = "Ajay". So TS is basically warning us that it "may" not work, but JS it is fine

so, fix it anyways

either change value to numeric or type to "any"

do not use "any" too much

### Built-in types

Boolean:

open types.ts

var isWinter : Boolean = false;

isWinter = 123;

compile: error??? No.

comment isWinter = 123;

var count: number = 5;

var fname: string = "Neo";

Array:

var names : string[] = ["Ajay", "Neo"];

This will give error:

var names: string[] = ["Ajay", "Neo", 5];

to store multiple types, define as "any": var names : any[] = ["Ajay", "Neo", 5];

Enum:

enum Starks { Jon, Bran, Heather, Catlyn };

var cat : Starks = Starks.Catlyn;

void:

function getName() : string {

return "Ajay"; // return 1 will throw an error.

}

use "void" when not returning anything:

function getName() : void {

console.log("some message");

}

### Interfaces

Interface is a defn for an object that tells TS what that obj is going to be

It is a blueprint for an object

in interfaces.ts:

function printName(stark) {

console.log(stark.name);

}

printName({name: "Ajay"});

printName({label: "Joe"});

compile: success. but run will give error.

run:

Ajay

undefined

difficult to determine what went wrong. Interface to the rescue.

in interfaces.ts:

interface Stark {

name: string;

}

function printName(stark: Stark) {

console.log(stark.name);

}

printName({name: "Ajay"});

printName({label: "Joe"});

Compile: error, "label" is missing in type.

change to:

printName({name: "Ajay"});

printName({name: "Joe"});

runs.

modify:

interface Stark {

name: string;

age: number;

}

compile: error. 2 errors. both names do not have property age, as it is "required".

to make age optional:

age?: number

compile: works

### Classes

create classes.ts

class Stark {

name: string;

}

var ned = new Stark();

compile. look at the .js code

ned.saying = "Winter is coming";

compile: error "property does not exist"

class Stark {

name: string;

saying: string;

}

// cannot have optional vars in classes

Classes can have ctors

class Stark {

name: string = "Brandon"; // default value.

saying: string;

static castle: string = "Winterfell";

constructor() {

this.saying = "Winterfell!";

}

}

console.log(Stark.castle);

compile + run: winterfell!

add methods to classes:

class Stark {

name: string = "Brandon"; // default value.

saying: string;

static castle: string = "Winterfell"!;

constructor() {

this.saying = "Winterfell!";

}

hello(person: string) {

console.log("Hello " + person);

}

}

//console.log(Stark.castle);

var ned = new Stark();

ned.hello("Robert");

### Inheritance

use "extends"

create extends.ts

class Person {

name: string;

constructor(name:string) {

this.name = name;

}

dance() {

console.log(this.name + " is dancing.");

}

}

var person = new Person("Bryan");

person.dance();

compile + run. works.

another class:

class AwesomePerson extends Person {

// override func.

dance() {

console.log("Sooo awesome!");

}

}

var robb = new AwesomePerson("Robb");

robb.dance();

compile+run:

Sooo awesome!

call base class' func:

class AwesomePerson extends Person {

// override func.

dance() {

console.log("Sooo awesome!");

super.dance(); // invoke func in base class.

}

}

var robb = new AwesomePerson("Robb");

robb.dance();

compile+run:

Sooo awesome!

Robb is dancing

### Implements

In Typescript an implements clause can be used to verify that a class conforms to a specific interface. If a class fails to implement an interface correctly, an error will be generated. Classes can implement a single interface or multiple interfaces at once.

**Example 1:**An interface A is created with has a function display of type void. Class B and Class C implement interface A. Class B doesn’t raise an error as method display is properly implemented but class C has a method Display(), which is different from the method defined in the interface, so typescript compiler raises an error that class ‘C’ incorrectly implements interface ‘A’.

// Creating an interface

interface A {

  display(): void;

}

class B implements A {

  display() {

    console.log("B");

  }

}

class C implements A {

  //  Throws error:  Class 'C' incorrectly

  // implements interface 'A'.

  //  Property 'display' is missing in type

  // 'C' but required in type 'A'.

  Display() {

    console.log("C");

  }

}

**Example 2:**If an additional property exists in an interface does not make that property exist in the class that implements it unless the properties are defined. In such cases typescript compiler raises errors.

interface A {

  propA: number;

  propB: number;

}

class B implements A {

  propA = 0;

}

const obj = **new** B();

obj.propB = 3;

### Static

ES6 includes static members and so does TypeScript. The static members of a class are accessed using the class name and dot notation, without creating an object e.g. <ClassName>.<StaticMember>.

The static members can be defined by using the keyword *static*.

The following example defines a class with static property and method and how to access it.

class Circle {

    static pi: number = 3.14;

    static calculateArea(radius:number) {

        return this.pi \* radius \* radius;

    }

    calculateCircumference(radius:number):number {

        return 2 \* Circle.pi \* radius;

    }

}

Circle.pi; // returns 3.14

console.log("Area of the circle is: " + Circle.calculateArea(5)); // returns 78.5

let circleObj = new Circle();

console.log("Circumference of the circle is: ", circleObj.calculateCircumference(5)); // returns 31.4000000

//circleObj.calculateArea(); <-- cannot call this

### Abstract

Define an abstract class in Typescript using the abstract keyword. Abstract classes are mainly for inheritance where other classes may derive from them. We cannot create an instance of an abstract class.

An abstract class typically includes one or more abstract methods or property declarations. The class which extends the abstract class must define all the abstract methods.

The following abstract class declares one abstract method find and also includes a normal method display.

abstract class Person {

    name: string;

    abstract city: string;

    constructor(name: string) {

        this.name = name;

    }

    display(): void{

        console.log(this.name, this.city);

    }

    abstract find(string): Person;

}

class Employee extends Person {

    empCode: number;

    city: string;

    constructor(name: string, code: number, city: string) {

        super(name); // must call super()

        this.empCode = code;

        this.city = city;

    }

    find(name:string): Person {

        // execute AJAX request to find an employee from a db

        return new Employee(name, 1, "Reston");

    }

}

let emp: Person = new Employee("James", 100, "Dallas");

emp.display(); //James

let emp2: Person = emp.find('Steve');

emp2.display();

In the above example, Person is an abstract class which includes one property and two methods, one of which is declared as abstract. The find() method is an abstract method and so must be defined in the derived class. The Employee class derives from the Person class and so it must define the find() method as abstract. The Employee class must implement all the abstract methods of the Person class, otherwise the compiler will show an error.

The abstract class can also include an abstract property. In the above example, there is city, which is an abstract property of the Person class.

### Modules

share code b/w files.

Create folder “modules”

create timesTwo.ts

function timesTwo(n:number) {

return n \* 2;

}

create new TS file, util.ts

console.log(timesTwo(9));

tsc util.ts

error. can't find timesTwo

have to define a dependency.

so, make timesTwo a module

module Utility {

export class Useful {

timesTwo(n:number) {

return n \* 2;

}

}

}

in util.ts:

/// <reference path="timesTwo.ts" />

var use = new Utility.Useful();

console.log(use.timesTwo(9));

compile: no error. Creates .js for both of them.

node util.js: error: Utility is not defined.

because we compiled only util.ts and not timesTwo.ts

We must combine both together and compile+run

TSC timesTwo.ts util.ts --out util.js

open util.js

node util.js

shows 18

* **More Module related information**

executed within their scope, not globally

unless exported and then imported in another module

Example #1: use the expClass and impClass ts files.

Another example: use the IShape, ICircle, ITraingle & TestShape ts files.

// MyClass.ts

#1

export class MyClass {

myFunction(x: string) {

return x;

}

}

#2:

class MyClass {

myFunction(x: string) {

return x;

}

}

export{MyClass};

// open another module file: importmodule.ts

import { MyClass } from "./MyClass"; // relative path

let object = new MyClass();

console.log(object.myFunction("value"));

#3: Export as a different name.

class MyClass {

myFunction(x: string) {

return x;

}

}

export{MyClass as MainClass};

// open another module file: importmodule.ts

import { MainClass } from "./MyClass"; // relative path

let object = new MainClass();

console.log(object.myFunction("value"));

### enums

enum keyword

enum Values {

First = 1,

Second = 2,

Third = 3,

Fourth = 4

}

let first = Values.First; // 1

let nameOfFirst = Values[Values.First]; // "First"

console.log(first)

console.log(nameOfFirst)

### Generics

function value(myval: number): number {

return myval;

}

function value2(myval2: any): any {

return myval2;

}

let val0 = value(6); // works

let val1 = value("test"); // compile error

let val2 = value2("test"); // works

let val3 = value2(24); // works

function type<T>(parameter: T): T {

return parameter;

}

function identity<T>(arg: T): T {

//console.log(arg.length); // Error. T doesn't have length.

return arg;

}

var output = identity<string>("Ajay");

console.log(output);

var output2 = identity("Ajay Singala");

console.log(output2);

// Array Generic

function identityLog<T>(arg: T[]): T[] {

console.log(arg.length); // Array has length. No error.

return arg;

}

var id = identityLog(["ajay", "neo", "trinity"]);

console.log(id[0]);

console.log(id[1]);

console.log(id[2]);

### Namespaces

internal modules => namespaces

external modules => modules

similar to C# to organize code

can be done in same file or separate files.

Create folder “namespaces”

// namespaces.ts file

namespace Primary {

export interface PrimaryInterface {

isTrue(x: string): boolean;

}

}

// importnamespaces.ts file

/// <reference path="namespaces.ts" />

namespace Primary {

export class SecondNamespace implements PrimaryInterface {

isTrue(x: string) {

if(x === "true") {

return true;

}

return false;

}

}

}

var sn = new Primary.SecondNamespace();

console.log(sn.isTrue(“true”));

console.log(sn.isTrue(“nope”));

### Iterators

var listItems = [5, 6, 7];

// show indexes.

for (var x1 in listItems) {

console.log(x1); // output: "0", "1", "2"

}

// show values.

for (var x2 in listItems) {

console.log(listItems[x2]); // output: 5, 6, 7

}

# Webpack

<https://levelup.gitconnected.com/what-is-webpack-4fdb624597ae>

<https://www.smashingmagazine.com/2021/06/getting-started-webpack/>

<https://v2.angular.io/docs/ts/latest/guide/webpack.html>

<https://medium.com/@sahayatanakul2867/configuration-of-webpack4-and-angular-8-using-two-approaches-2bbc4b1cacda>

<https://www.imperva.com/learn/performance/minification/>

# SPA Definition

<https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application>

Single page applications (SPAs) are everywhere. Even if you're not exactly sure what they are, you most likely use them regularly — they're a great tool for making incredibly engaging and unique experiences for website users.

A single page application is a website or web application that dynamically rewrites a current web page with new data from the web server, instead of the default method of a web browser loading entire new pages.

You'll easily recognize some popular examples of single page applications like Gmail, Google Maps, Airbnb, Netflix, Pinterest, Paypal, and many more. Companies all over the internet are using SPAs to build a fluid, scalable experience.

### What Is a Single Page Application?

A single page application is a single page (hence the name) where a lot of information stays the same and only a few pieces need to be updated at a time.

For example, when you browse through your email, you’ll notice that not much changes during navigation — the sidebar and header remain untouched as you go through your inbox.

The SPA only sends what you need with each click, and your browser renders that information. This is different than a traditional page load where the server rerenders a full page with every click you make and sends it to your browser.

This piece-by-piece, client-side method makes load times much faster for users. It also lessens the amount of information a server has to send and makes the whole process a lot more cost-efficient — a win-win scenario for users and businesses.

### What Is Single Page Application Architecture? How Does It Work?

It's obvious to see the benefits of a single page application. Utilizing a web application or website that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server, provides a much better user experience.

It avoids interruptions in a user's journey, which is vital for websites — especially in [digital commerce](https://www.bloomreach.com/en/blog/2019/digital-commerce-explained). By cutting down on lag time between successive pages, it makes the site behave more like a desktop application, providing a more fluid and comfortable experience.

And this can make a huge impact. Because on most websites, there is a lot of repeating content.

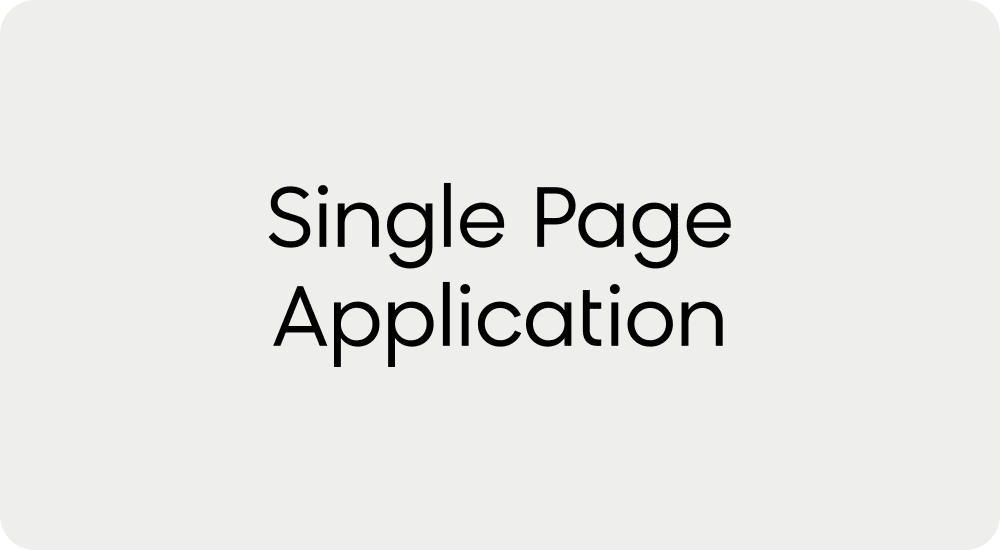
Some of it stays the same no matter where the user goes (headers, footers, logos, and the navigation bar), while some of it is constant in just a certain section (filter bars and banners). And there are many repeating layouts and templates (blogs, self-service pages, or the aforementioned Google mail setup).

**Single page applications take advantage of this repetition.**

Let’s say you visit a website and view a painting of a house and a tree. Traditional multi-page websites paint the entire picture for you on the server and send it over to your browser.



An SPA gives you the paint-by-numbers guide for the site, including the repeating guides you’ll likely be using, and then pipes in the right paint (data and content) to fill out the template.



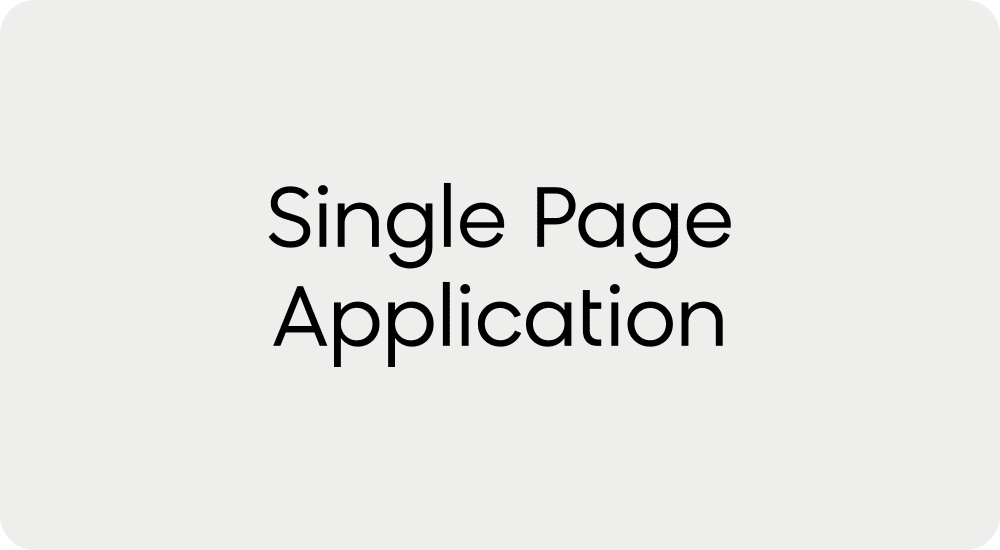
Either way, you see the same tree. But the speed of single page applications comes in when you request new content — like clicking on “next,” filtering results, opening an email, or in our case, asking to see a different tree.

On a traditional website, your request for a new tree would cause the server to repaint the entire picture and send it back.



With a single page application, the server says, “Hey, I’ve got a new tree for you, but you’ve already got the house so just leave that the same.” Then, it sends the page updated instructions for a new tree and the paint to make it.

By transferring the painting work (or page rendering) from the server to the client (you), the page can be dynamically rewritten instead of going through an entire reload. **This makes things a whole lot faster.**



### Advantages of Single Page Applications

There are many benefits to SPA solutions on both the [customer experience](https://www.bloomreach.com/en/blog/2020/ecommerce-customer-experience) side and back-end design side of the equation. Improved application performance, consistency, reduced development time, and lower infrastructure costs not only help you offer a more pleasant experience to your users, they also help development teams operate more effectively.

By separating the presentation from the content and data, development teams can work at different speeds while still remaining integrated and working towards the overall solution. SPAs are also good for making responsive designs for mobile, desktop, and tablet.

#### **[Advantage #1]** A **Single-time File Load Each For HTML, CSS, JS**

With a single page application, after the initial page load, the server doesn’t send any more HTML to you —  you download it all right at the beginning. The server sends you a shell page and your browser renders the user interface (UI).

Then, as you click around, the SPA sends back requests for data and markup. The server shoots back the raw materials needed, and your browser takes it and renders an updated UI, interchanging pieces without ever needing to refresh the full page. This quick interchangeability makes SPAs incredibly useful on pages that are highly navigated and use repeating templates.

#### **[Advantage #2] No Extra Queries to Server**

Because the server doesn’t need to spend time and energy doing multiple renderings, SPAs lower the impact on your servers overall, meaning you can save money by using fewer servers for the same amount of traffic.

#### **[Advantage #3]  Fast and Responsive Front-end Building**

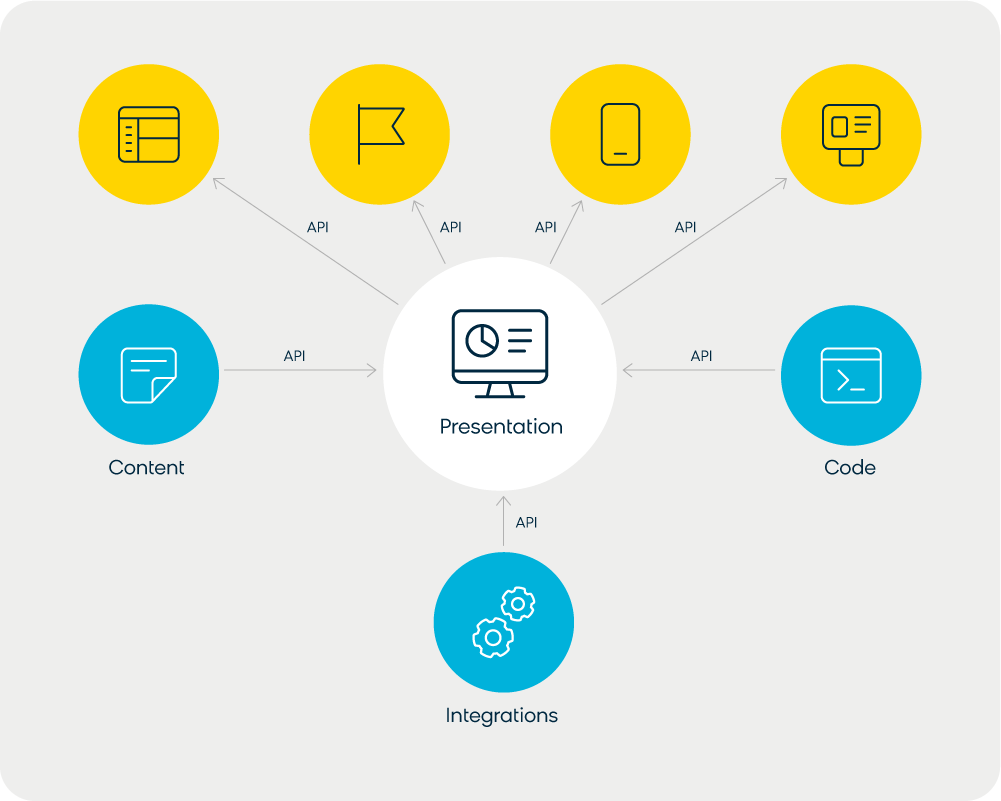
Along with a quicker performance time, SPAs also let developers build the front end of sites faster. This is due to the decoupled architecture of SPAs, or a separation of back-end services and front-end display.

Many business-critical functionalities don’t change all that much on the back end. How your customers log in, register, purchase, and track orders may change its “look” or presentation from time to time, but the logic and data orchestration behind it is pretty constant — and you don’t want to risk messing that up.

Similarly, your raw content and data might stay the same, but the way you want to display it may differ. By decoupling that back-end logic and data from how it’s presented, you turn it into a “service,” and developers can build many different front-end ways to show and use that service.

With a [decoupled setup](https://www.bloomreach.com/en/blog/2019/headless-commerce), developers can build, deploy, and experiment with the front end completely independent from the underlying back-end technology. They design how they want the user experience to look and feel, and then pull in the content, data, and functionality through those services.

This is done using APIs, which are a standard set of rules between applications on how they will structure, exchange, and reassemble data.



This API setup lets developers work quickly on the UI with no risk to business-critical back-end technologies.

#### **[Advantage #4] Enhanced User Experiences**

As more and more functionalities are built as modular services (a microservice architecture) that can be updated independently, it becomes easier to experiment with how they are displayed and used.

SPA frameworks are great for trialing these services to create engaging, dynamic, and even animated user experiences**.**

Plus, many people simply prefer developing in a certain programming language (many SPA frameworks use JavaScript) and thanks to APIs, the SPAs you build in one language can work seamlessly with back-end services developed in different languages.

# Angular

<https://www.simplilearn.com/tutorials/angular-tutorial>

<https://www.tektutorialshub.com/angular-tutorial/>

<https://angular.io/tutorial>

## History

Following are the Angular version release dates:

* Angular version 1.0, which is known as AngularJS, was released in 2010 by Google
* Angular version 2.0 was released in September 2016
* Angular 4.0 was released in March 2017
* Angular 5.0 was released in Nov 2017
* Angular 6.0 was released in May 2018
* Angular 7.0 was released in Oct 2018
* Angular 8.0 was released in May 2019
* Angular 9.0 was released in Feb 2020
* Angular 10.0 was released in June 2020
* Angular 11.0 was released in Nov 2020
* Angular 12 is released in May 2021
* Angular 13 is released in Nov 2021
* Angular 14 is released in June 2022

## What Is Angular?

Angular is an open-source, [JavaScript](https://www.simplilearn.com/tutorials/javascript-tutorial/introduction-to-javascript) framework written in [TypeScript](https://www.simplilearn.com/tutorials/typescript-tutorial/typescript-interview-questions). Google maintains it, and its primary purpose is to develop single-page applications. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with. It enables users to create large applications in a maintainable manner.

## Why Do You Need a Framework?

Frameworks in general boost web development efficiency and performance by providing a consistent structure so that [developers](https://www.simplilearn.com/how-to-become-complete-web-development-professional-article) don’t have to keep rebuilding code from scratch. Frameworks are time savers that offer developers a host of extra features that can be added to software without requiring extra effort.

[JavaScript](https://www.simplilearn.com/reasons-to-learn-javascript-article) is the most commonly used client-side scripting language. It is written into [HTML](https://www.simplilearn.com/tutorials/html-tutorial/what-is-html) documents to enable interactions with web pages in many unique ways. As a relatively easy-to-learn language with pervasive support, it is well-suited to develop modern applications.

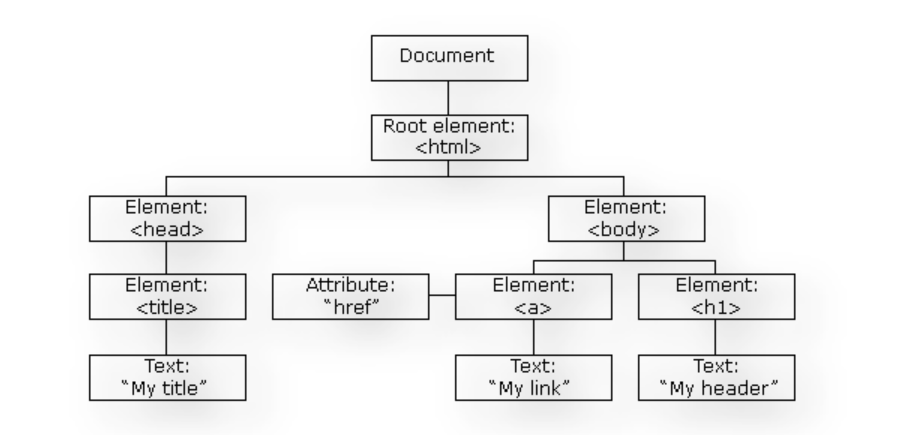
But is JavaScript ideal for developing single-page applications that require modularity, testability, and developer productivity? Perhaps not.

These days, we have a variety of frameworks and libraries designed to provide alternative solutions. With respect to [front-end web development,](https://www.simplilearn.com/how-to-become-a-front-end-developer-article) Angular addresses many, if not all, of the issues developers face when using JavaScript on its own.

## Features of Angular

### 1. Document Object Model

DOM (Document Object Model) treats an [XML](https://www.simplilearn.com/tutorials/programming-tutorial/what-is-xml) or HTML document as a tree structure in which each node represents a part of the document.



Angular uses regular DOM. Consider that ten updates are made on the same HTML page. Instead of updating the ones that were already updated, Angular will update the entire tree structure of [HTML tags](https://www.simplilearn.com/tutorials/html-tutorial/html-tags).

### 2. TypeScript

TypeScript defines a set of types to JavaScript, which helps users write JavaScript code that is easier to understand. All of the TypeScript code compiles with JavaScript and can run smoothly on any platform. TypeScript is not compulsory for developing an Angular application. However, it is highly recommended as it offers better syntactic structure—while making the codebase easier to understand and maintain.



You can install TypeScript as an NPM package with the following command:

npm install -g typescript

### 3. Data Binding

[Data binding](https://www.simplilearn.com/tutorials/angular-tutorial/angular-data-binding) is a process that enables users to manipulate web page elements through a web browser. It employs dynamic HTML and does not require complex scripting or [programming](https://www.simplilearn.com/how-to-learn-programming-article). Data binding is used in web pages that include interactive components, such as calculators, tutorials, forums, and games. It also enables a better incremental display of a web page when pages contain a large amount of data.

Angular uses the two-way binding. The model state reflects any changes made in the corresponding UI elements. Conversely, the UI state reflects any changes in the model state. This feature enables the framework to connect the DOM to the model data through the controller.

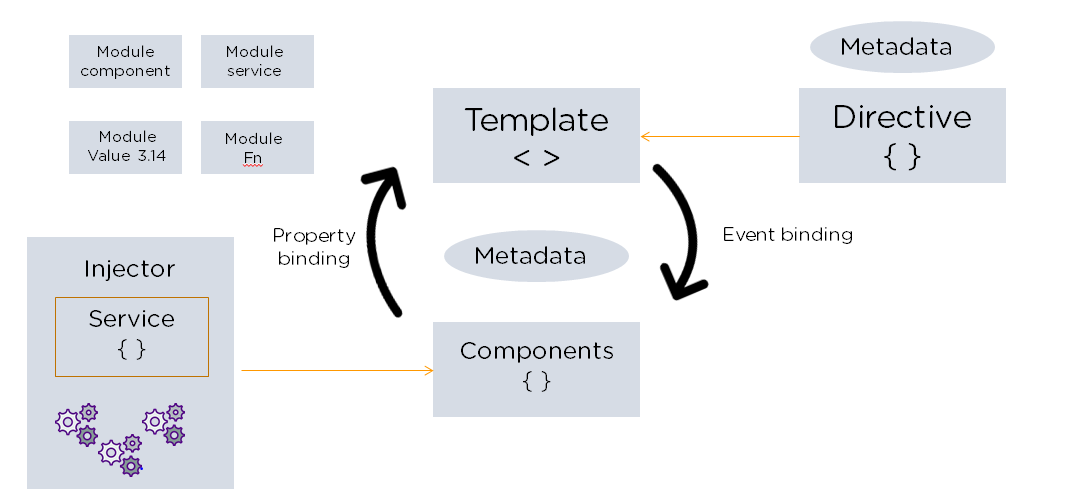
### 4. Testing



Angular uses the [Jasmine testing framework](https://en.wikipedia.org/wiki/Jasmine_(JavaScript_testing_framework)). The Jasmine framework provides multiple functionalities to write different kinds of test cases. Karma is the task-runner for the tests that uses a configuration file to set the start-up, reporters, and testing framework.

## Angular Architecture

Angular is a full-fledged [model-view-controller (MVC) framework.](https://www.simplilearn.com/tutorials/dot-net-tutorial/mvc-architecture) It provides clear guidance on how the application should be structured and offers bi-directional data flow while providing real DOM.



The following are the eight building blocks of an Angular application:

### 1. Modules

An Angular app has a root module, named AppModule, which provides the bootstrap mechanism to launch the application.

### 2. Components

Each [component](https://www.simplilearn.com/tutorials/angular-tutorial/angular-components) in the application defines a class that holds the application logic and data. A component generally defines a part of the user interface (UI).

### 3. Templates

The Angular template combines the Angular markup with HTML to modify HTML elements before they are displayed. There are two types of data binding:

1. Event binding: Lets your app respond to user input in the target environment by updating your application data.
2. Property binding: Enables users to interpolate values that are computed from your application data into the HTML.

### 4. Metadata

Metadata tells Angular how to process a class. It is used to decorate the class so that it can configure the expected behavior of a class.

### 5. Services

When you have data or logic that isn’t associated with the view but has to be shared across components, a [service](https://www.simplilearn.com/tutorials/angular-tutorial/angular-service) class is created. The class is always associated with the @Injectible decorator.

### 6. Dependency Injection

[This feature](https://www.simplilearn.com/tutorials/angular-tutorial/angular-dependency-injection) lets you keep your component classes crisp and efficient. It does not fetch data from a server, validate the user input, or log directly to the console. Instead, it delegates such tasks to the services.

Angular comes with its own set of advantages and disadvantages. The next two sections briefly explain them.

## AngularJS Directives

AngularJS directives extend the HTML by providing it with new syntax. You can easily spot directives because they have the prefix “ng-.” Consider them markers on the DOM element, instructing AngularJS to attach a certain behavior to the element, or even change it outright.

Here are two sample directives:

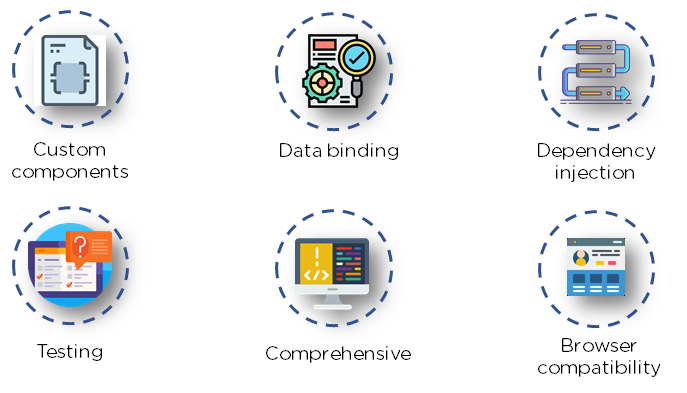
* **The ng-model Directive**

The ng-model binds the value of the HTML control with the specified AngularJS expression value.

* **The ng-bind Directive**

This directive replaces the HTML control value with a specified AngularJS expression value.

## Advantages of Angular



Many versions of Angular have been released since its inception. All these versions have added to the efficient working of the framework.

### 1. Custom Components

Angular enables users to build their own components that can pack functionality along with rendering logic into reusable pieces. It also plays well with web components.

### 2. Data Binding

Angular enables users to effortlessly move data from JavaScript code to the view, and react to user events without having to write any code manually.

### 3. Dependency Injection

Angular enables users to write modular services and inject them wherever they are needed. This improves the testability and reusability of the same services.

### 4. Testing

Tests are first-class tools, and Angular has been built from the ground up with testability in mind. You will have the ability to test every part of your application—which is highly recommended.

### 5. Comprehensive

Angular is a full-fledged framework and provides out-of-the-box solutions for server communication, routing within your application, and more.

### 6. Browser Compatibility

Angular is cross-platform and compatible with multiple browsers. An Angular application can typically run on all browsers (Eg: Chrome, Firefox) and OSes, such as Windows, macOS, and Linux.

## Limitations of Angular



### 1. Steep Learning Curve

The basic components of Angular that all users should know include directives, modules, decorators, components, services, dependency injection, [pipes](https://www.simplilearn.com/tutorials/angular-tutorial/angular-pipes), and templates. More advanced topics include change detection, zones, AoT compilation, and Rx.js. For beginners, Angular 4 may be challenging to learn because it is a complete framework.

### 2. Limited SEO Options

Angular offers limited SEO options and poor accessibility to search engine crawlers.

### 3. Migration

One of the reasons why companies do not frequently use Angular is the difficulty in porting legacy js/jquery-based code to angular style architecture. Also, each new release can be troublesome to upgrade, and several of them are not backward-compatible.

### 4. Verbose and Complex

A common issue in the Angular community is the verbosity of the framework. It is also fairly complex compared to other front-end tools.

## Angular vs AngularJS

| **Key** | **AngularJS** | **Angular** |
| --- | --- | --- |
| **Architecture** | AngularJS works on MVC, Model View Controller Design. Here View shows the information present in the model and controller processes the information. | Angular uses components and directives. Here component is directive with a template. |
| **Language** | AngularJS code is written in javascript. | Angular code is written in typescript. |
| **Mobile** | AngularJS code is not mobile friendly. | Angular develped applications are mobile browser friendly. |
| **Expression syntax** | {{}} are used to bind data between view and model. Special methods, ng-bind can also be used to do the same. | () and [] attributes are used to bind data between view and model. |
| **Dependency Injection** | DI is not used. | Hierarchical DI system is used in Angular. |
| **Routing** | @routeProvider.when, then are used to provide routing information. | @Route configuration is used to define routing information. |
| **Management** | AngularJS project is difficult to manage with increasing size of the source code. | Angular code is better structured, is easy to create and manage bigger applications. |

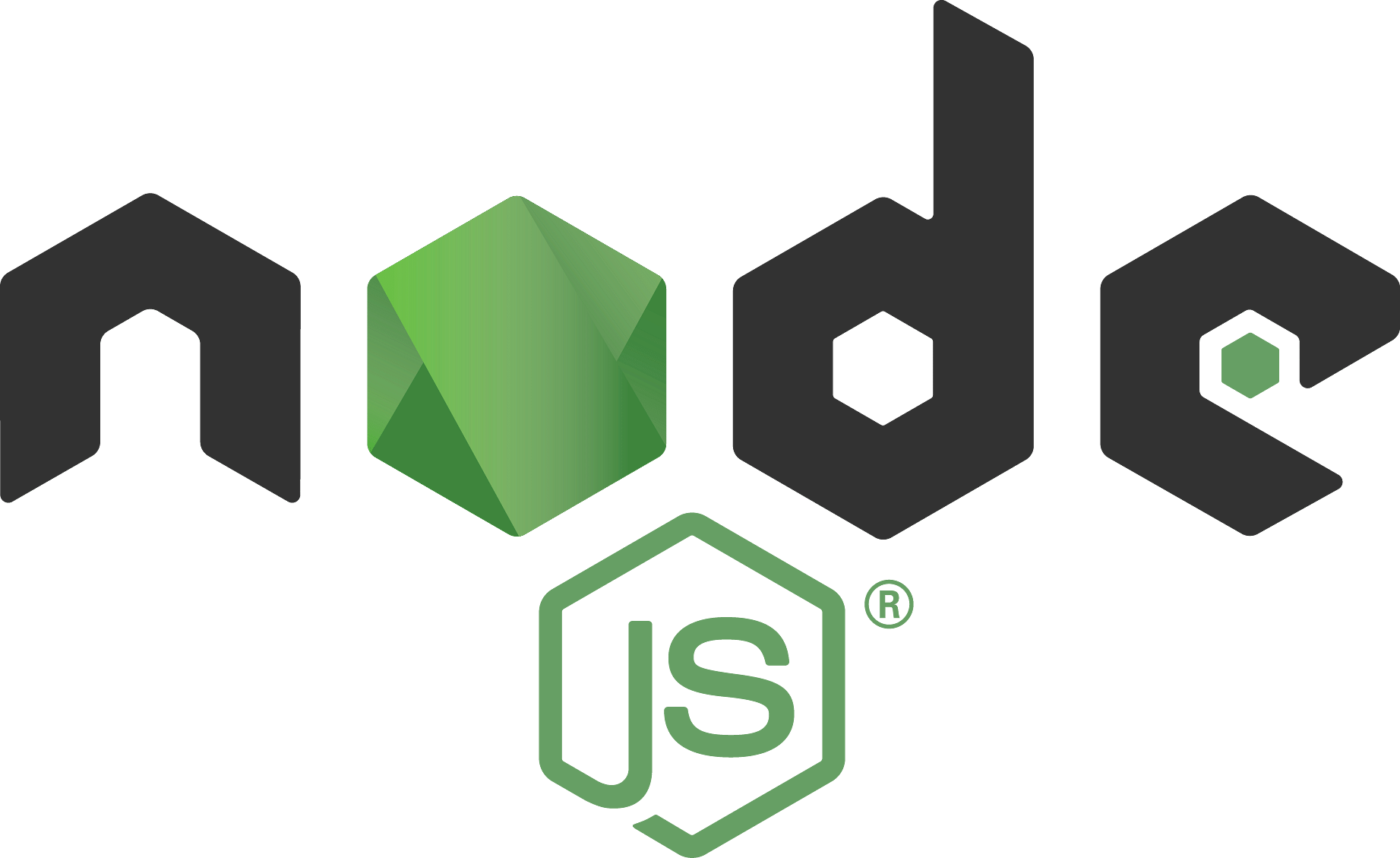
# Hello World – First Angular App

**All code in /Angular/Angular Demos**

## Angular Prerequisites

There are three main prerequisites.

### NodeJS



Angular uses [Node.js](https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs) for a large part of its build environment. As a result, to get started with Angular, you will need to have Node.js installed on your system. You can head to the NodeJS official website to download the software. Install the latest version and confirm them on you command prompt by running the following commands:

**node --version**

**npm --v**

### Angular CLI



The Angular team has created a command-line interface (CLI) tool to make it easier to bootstrap and develop your Angular applications. As it significantly helps to make the process of development easier, we highly recommend using it for your initial [angular projects](https://www.simplilearn.com/tutorials/angular-tutorial/angular-project) at the least.

To install the CLI, in the command prompt, type the following commands

**Installation**:

**npm install -g @angular/cli**

**Confirmation**:

**ng version**

### Text Editor



You need a text editor to write and run your code. The most popularly used integrated development environment (IDE) is Visual Studio Code (VS Code). It is a powerful source code editor that is available on Windows, macOS, and Linux platforms.

## Creating the Angular HelloWorld Application

### **Step 1**

Create a folder for your application in the desired location on your system and open it on VSCode. Open a new terminal and type in the following command to create your app folder (m*ake sure you navigate to a new folder for your demos first and then create your new Angular projects*).

**ng new hello-world**

When the command is run, Angular creates a skeleton application under the folder. It also includes a bunch of files and other important necessities for the application.

### **Step 2**

To run the application, change the directory to the folder created, and use the ng command.

**cd hello-world**

**ng serve**

Once run, open your browser and navigate to localhost:4200. If another application is running on that address, you can simply run the command.

**ng serve --port**

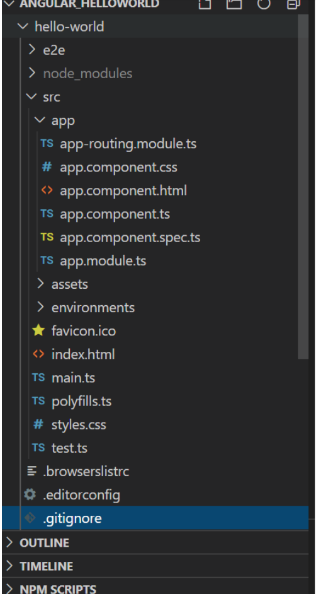
It will generate a port for you to navigate to. OR provide a specific port number:

**ng serve --port 4300**

You can leave the ng serve command running in the terminal and continue making changes. If you have the application opened in your browser, it will automatically refresh each time you save your changes. This makes the development quick and iterative.

### Basics of an Angular App

At its core, any Angular application is still a Single-Page Application (SPA), and thus its loading is triggered by a main request to the server. When we open any URL in our browser, the very first request is made to our server. This initial request is satisfied by an HTML page, which then loads the necessary JavaScript files to load both Angular as well as our application code and templates.



### Root HTML - index.html

The root component looks very pristine and neat, with barely any references or dependencies. The only main thing in this file is the **<app-root>** element. This is the marker for loading the application. All the application code, styles, and inline templates are dynamically injected into the index.html file at run time by the **ng serve** command.

### The Entry Point - main.ts

The second important part of our bootstrapping piece is the main.ts file. The index.html file is responsible for deciding which files are to be loaded. The main.ts file, on the other hand, identifies which Angular module is to be loaded when the application starts.

Most of the code in the main.ts file is generic, and you will rarely have to touch or change this entry point file. Its main aim is to point the Angular framework at the core module of your application and let it trigger the rest of your application source code from that point.

### Main Module - app.module.ts

This is where your application-specific source code starts from. The application module file can be thought of as the core configuration of your application, from loading all the relevant and necessary dependencies, declaring which components will be used within your application, to marking which is the main entry point component of your application.

### Root Component – AppComponent – app.component.ts

The app.component.ts is the actual Angular code that drives the functionality of the application.

A component in Angular is nothing but a TypeScript class, decorated with some attributes and metadata. The class encapsulates all the data and functionality of the component, while the decorator, @Component, specifies how it translates into the HTML.

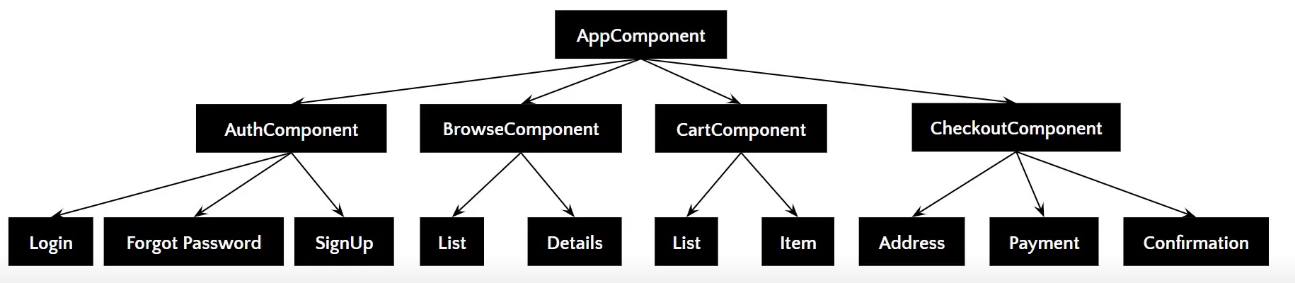
We suggest you check out the article on Angular components to understand the concept better.

Back in our app.component.html file of the Angular HelloWorld application, you can delete all the code and type in something that you wish to render on the browser.

# Angular Components

## What Are Angular Components?

Components are the building blocks of a UI in an Angular application. These components are associated with a template and are a subset of directives.



The above image gives the tree structure of classification. There’s a root component, which is the AppComponent, that then branches out into other components creating a hierarchy.

Here are some of the features of Angular Component:

* Components are typically custom HTML elements, and each of these elements can instantiate only one component.
* A TypeScript class is used to create a component. This class is then decorated with the “@Component” decorator.
* The decorator accepts a metadata object that gives information about the component.
* A component must belong to the NgModule in order for it to be usable by another component or application.
* Components control their runtime behavior by implementing Life-Cycle hooks.

import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'hello-world';

}

The above code shows an App component, which is a pure TypeScript class decorated with the “@Component” decorator. The metadata object provides properties like selector, templateUrl, and so on—the templateUrL points to an HTML file that defines what you see on your application.

In the index.html file, <app-root> tag corresponds to component’s selector. By doing so, Angular will inject the corresponding template of the component.

<body>

  <app-root></app-root>

</body>

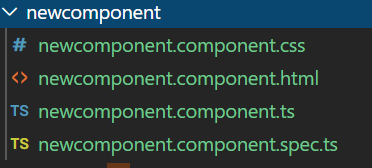
</html>

## Creating Your First Angular Component

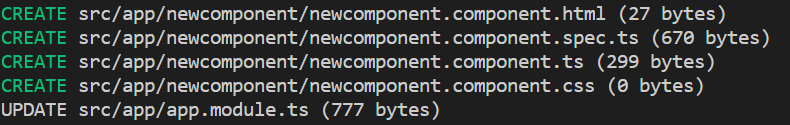
* To create an Angular Component, Angular CLI is used. In the terminal, type in the command,

  ng g c component-name

* This will create a folder named component-name with four files.



You’ll also receive a message:



The update message indicates that the component created is included in the declarations array of the main component.

It is crucial for Angular to know which component is to be run next and its features. For that, some metadata is created. The next section addresses the component metadata.

**To generate without the .spec.ts:**

ng generate component --skip-tests=true component-name

For a single project, change or add the following in your angular.json:

{

"projects": {

"{PROJECT\_NAME}": {

"schematics": {

"@schematics/angular:component": {

"skipTests": true

}

}

}

}

}

## Component Decorator Metadata

As mentioned earlier, the @Component decorator accepts a metadata object that provides information about the component. Here’s a list of properties of the metadata object:

@Component({

  selector: 'app-root',

  template: `<h1>Hello! Welcome</h1>`,

  templateUrl: './app.component.html',

  styles: [`

    h3{

      color: blue;

    }

  `],

  styleUrls: ['./app.component.css']

### Selector

It is the CSS selector that identifies this component in a template. This corresponds to the HTML tag that is included in the parent component. You can create your own HTML tag. However, the same has to be included in the parent component.

### Template

It is an inline-defined template for the view. The template can be used to define some markup. The markup could typically include some headings or paragraphs that are displayed on the UI.

### TemplateUrl

It is the URL for the external file containing the template for the view.

### Styles

These are inline-defined styles to be applied to the component’s view

### styleUrls

List of URLs to stylesheets to be applied to the component’s view.

### Providers

It is an array where certain services can be registered for the component

### Animations

Animations can be listed for the components

## Demo: Creating an Angular Component

Now that you have a good understanding of Angular components let me help you create an application using them.

**Step 1**: First, create a folder in your application to store all your components in.

ng g c components/new-component

Observe that the extension .component is appended to indicate that it is indeed a component.

**Step 2**: Within the component, open the new-component.component.html file to type in whatever you’d like to see on the browser.

<h1>Hey! I'm the first component</h1>

**Step 3**: In the new-component.component.ts file, copy the selector property to incorporate it in the app.component.html file.

@Component({

  selector: 'app-new-component',

  templateUrl: './new-component.component.html',

  styleUrls: ['./new-component.component.css']

})

In the app.component.html, which is the root component, go ahead and define the custom HTML tag. This indicates that the component created is being incorporated for the final render.

<h1>Welcome to this tutorial on Angular Components</h1>

<app-new-component></app-new-component>

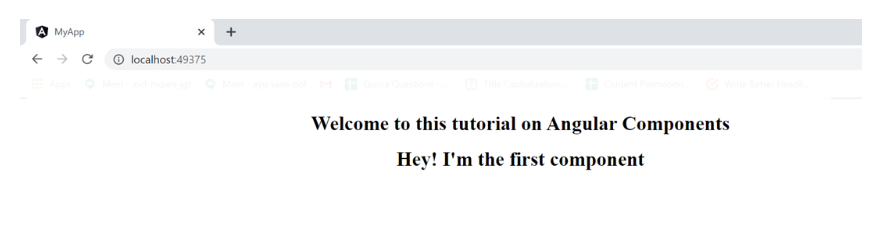
You can also define any styling conventions for the component in the CSS file.

h1 {

    text-align: center;

}

Once you execute the ng serve command, the output looks something like this.



You can create multiple components and define the tags in the app component. The components are executed sequentially.

Here, I’ve created another component within which I’ve embedded the Angular logo. Create folder src/assets and copy the image in this folder.

<h1>This is the image component</h1>

<img src= "assets/Angular\_logo.png" class="center" style="height: 100px; width: 100px">

I’ve also specified styling conventions in its corresponding CSS file.

h1 {

    color: green;

    text-align: center;

}

.center {

    display: block;

    margin-left: auto;

    margin-right: auto;

    width: 50%;

  }

View the page again. It will auto-refresh.

# Angular – Data Binding

## What Is Angular Data Binding?

Data binding allows Internet users to manipulate web page elements with the help of a web browser. It includes dynamic HTML and does not require complex programming. Data binding is used in web applications that contain interactive components such as forms, calculators, tutorials, and games. The incremental display of a webpage makes data binding convenient when pages contain an extensive amount of data.

Angular uses the concept of two-way binding. Any UI element-related change is reflected in the corresponding and specific model state. Conversely, any model state changes reflect in the UI state. This ensures that the framework is able to connect the DOM to the Model data with the help of the controller.

## AngularJS Controller

The AngularJS Controller is the part of the AngularJS application that controls the data binding. It is responsible for linking the data to the view and vice versa. The Controller also handles all the user input and output.

The AngularJS Controller is very important in AngularJS applications as it is responsible for managing the data binding. Without a controller, the data binding would not be possible, as the user would not be able to interact with the application.

## Attribute Binding

Attribute binding in Angular helps you bind to HTML attributes of elements in your template. This can be useful when you want to dynamically update the appearance or behavior of an element based on some condition. For example, you might want to hide an element unless a user is logged in, or change the color of an element based on its status. To bind to an attribute, you use the square brackets around the attribute name.

## Class Binding

Angular offers various ways to bind data to HTML elements. Class binding is one of them. It allows you to dynamically add or remove CSS classes from an element. This can be useful for applying styles based on certain conditions.

## Style Binding

Style binding is a one-way data binding technique that can be used to set the value of a CSS property on an element. To use style binding, you first need to have a CSS property that you want to bind to an element.

## ngModel

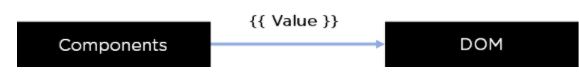
Angular data binding is a two-way process: it can both send and receive data. This means that when you change something in your view, the model updates automatically, and vice versa. The ngModel directive makes this two-way data binding possible.

When you use the ngModel directive, you specify a property of the scope as the value of the directive. This tells Angular to create a two-way binding between the property and the input control. Any changes to the control are automatically reflected in the model, and any changes to the model are automatically reflected in the control.

## Types of Data Binding

### Interpolation Binding

Interpolation is a procedure that allows the user to bind a value to the user interface element. Interpolation binds the data one-way, which means that data moves in one direction from the components to HTML elements.



We’ve added the code for the same below.

In the app.component.ts file, we’ve created two properties called name and topic.

export class AppComponent {

  title = 'binding';

  public name = "Simplilearn"

  public topic = "Data Binding"

To interpolate and bind them in the HTML file, type the following code

<h1 style="text-align: center;">Welcome to {{name}}</h1>

<h2 style="text-align: center;">Welcome to the {{topic}} tutorial</h2>

The output will look like this:



### Property Binding

Property binding is a one-way data binding mechanism that allows you to set the properties for HTML elements. It involves updating a property value in the [component](https://www.simplilearn.com/tutorials/angular-tutorial/angular-components) and binding the value to an HTML element in the same view. We use property binding for toggle functionality and sharing data between components. It uses the "[]" syntax for data binding.

In the app.component.ts file, I’ve created another property called image and provided the path to the Logo in the assets folder.

public image = "/assets/Logo.png"

In the app.component.html file.

<img [src] = "image" alt="" style="height: 100px; width: 250px" class="center">

The output now looks like this.



### Event Binding

Event binding type is when information flows from the view to the component when an event is triggered. The event could be a mouse click or keypress. The view sends the data to update the component. Unsurprisingly, it is the exact opposite of property binding, where the data goes from the component to the view.

view-event

We have created a Subscribe button that displays a “Thank you” message when clicked on.

<br><button (click)="onClick()">Subscribe to Simplilearn</button></div>

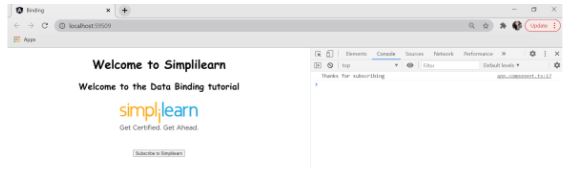
To display the message on the console, we’ve created a function called onClick() in the app.component.ts file.

onClick(){

    console.log("Thanks for subscribing")

  }

Every time the user clicks on the button, the message is displayed on the console.



### Two-way Data Binding

As the name suggests, two-way binding is a mechanism where data flows from the component to the view and back. This binding ensures that the component and view are always in sync. Any changes made on either end are immediately reflected on both. The general syntax to denote two-way data binding is a combination of Square brackets and parentheses "[()]".

2-way

To illustrate two-way data binding, we’ve created a property with an empty string and an input box for the user to type. Whatever the user provides is displayed on the screen with the help of the property.

In app.module.ts:

import { FormsModule } from '@angular/forms';

and

imports: [

BrowserModule,

FormsModule

],

In the app.component.ts file,

public random = ""

In the app.component.html file, we’ve created an input field

<input [(ngModel)]="random" type="text"> <br>

  {{random}}

We have used the ngModel directive and initialized it to random. We’ve then interpolated the property random.

Check the refreshed page.

As you can see, it stores the text in the input field in the random property.

Change the onClick() method in app.component.ts:

onClick() {

    console.log("Thanks for subscribing.")

    this.random = "R E S E T!"

  }

Now, on the refreshed page, enter some text in the box and then, click on the “Subscribe” button to see the result.

# Angular – Components – Input and Output

Diagram

Description automatically generated

Angular Framework is built upon small components so passing the data from Parent component to child component is tricky. In that scenario, **@Input** and **@Output** Decorator comes in handy. You can define the properties of components you create and make them available in your whole application.

Angular components have a better way of notifying parent components that something has changed via events. **For example, inputs** specify which properties you can set on a component from a parent, whereas “**Outputs”** identifies the events a component can fire to send information up the hierarchy to its parent from its child component.

You can define the properties for components you create and make them available across the whole angular app. In this Angular 12 Input Output tutorial, we will display the parent component’s data into a child component and the child component’s data into the parent component.

## Angular @Input

Angular input decorator is just telling Angular, hey, when you find a property binding with this name, map it to my component property of this other name. Or, if I don’t give you an alias, use my component property name.

Now, even though Angular supports this name alias, it is a recommended practice to avoid using that approach by default. And try and use the class property name instead.

The Decorator marks the class field as an input property and supplies configuration metadata.

The input property is bound to the DOM property in the template. Therefore, during change detection, Angular automatically updates a data property with a DOM property’s value.

## Angular @Output

The Decorator that marks a class field as an output property and supplies configuration metadata. The DOM property bound to an output property is automatically updated during change detection.

You can supply the optional name to use in the templates when a component is instantiated that maps to the name of the bound property. By default, the original name of the required property is used for output binding.

**~~Source~~**~~: Angular Demos/inout~~

## **~~Step 1~~**~~: Create a new project or use an existing one.~~

~~Create a new project with ng new inout.~~

## ~~Step 2: Create parent and child components.~~

~~Go to the terminal and type the following command.~~

~~ng g~~ **~~c~~** ~~parent  
ng g~~ **~~c~~** ~~child~~

~~So, it will create an individual folder. Type the following command to start the Angular development server.~~

~~ng serve --open~~

~~It will open up the browser at the~~**~~port: 4200~~**~~.~~

~~Right now, only the~~**~~app.component.ts~~**~~component is rendered in the browser. If we want to render our parent component, we need to include it in an~~**~~app.component.html~~**~~file as an HTML tag. Open~~ **~~app.component.html~~** ~~and replace its contents with this:~~

~~<div style="text-align:center">  
 <h1>  
 Welcome to {{ title }}!  
 </h1>  
 <app-parent></app-parent>  
</div>~~

~~Now, if you see in the browser, you can see the parent component renders. “~~**~~parent works!!~~**~~“~~

## **~~Step 3~~**~~: Define HTML for parent component.~~

~~Write the following code in the~~**~~parent.component.html~~**~~file.~~

~~<h3>Parent Component</h3>  
  
<label>Bitcoin price </label>  
<input type="text" />  
  
<p>Value of child component is: </p>~~

~~First, we pass the data from the parent component to the child component. Here is the scenario, when the user types the bitcoin price in the text box, we can see its worth in the child component.~~

~~The same scenario applies to the child component. When the child component starts entering the price, it will display in the parent component.~~

## ~~Step 4: Define HTML for child components.~~

~~Write the following code in the~~**~~child.component.html~~**~~file.~~

~~<h3>Child~~ **~~Component~~**~~</h3>  
  
<~~**~~label~~**~~>Child~~ **~~Component~~** ~~</~~**~~label~~**~~>  
<input~~ **~~type~~**~~="text" />  
  
<p>Value~~ **~~of~~** ~~parent~~ **~~component~~****~~is~~**~~: </p>~~

~~As we know, this is a child component, so we need to include the~~**~~<app-child>~~**~~tag into the parent component. So, our parent component HTML looks like this.~~

~~<h3>Parent Component</h3>  
  
<label>Parent Component</label>  
<input type="text" />  
  
<p>Value of child component is: </p>  
<app-child></app-child>~~

~~So, our application looks like this.~~

~~Graphical user interface, application

Description automatically generated~~

## ~~Step 5: Use Input to display parent component value~~

~~Create a reference to the input text of the parent component. To edit the following lines in the~~**~~parent.component.html~~**~~file.~~

~~<~~**~~input~~** ~~type="text"~~ **~~#pcomponent~~** ~~(keyup)="0"/>  
  
<app-child [PData]="pcomponent.value"></app-child>~~

~~First, I have defined the reference for the input tag and then set the event listener. Then, when a user types something in the textbox, it will pass the value as a property to the child component.~~

~~The child component is ready to receive the property via the~~**~~@Input Decorator~~**~~. So, this is the first use case of Inputs in Angular.~~

~~File~~**~~child.component.ts~~**~~file looks like this.~~

*~~// child.component.ts~~***~~import~~** ~~{ Component, OnInit, Input }~~ **~~from~~** ~~'@angular/core';~~**~~@Component~~**~~({  
 selector: 'app-child',  
 templateUrl: './child.component.html',  
 styleUrls: ['./child.component.css']  
})~~**~~export~~****~~class~~** ~~ChildComponent~~ **~~implements~~** ~~OnInit {~~ **~~@Input~~**~~() PData: string = “0”;~~ **~~constructor~~**~~() { }  
  
 ngOnInit() {  
 }  
  
}~~

~~You can see this component’s property is PData, which is the same property we have written in the~~**~~parent.component.html~~**~~file.~~

~~Finally, our~~**~~child.component.html~~**~~file looks like this. We need to add interpolation to display the parent data in the child component.~~

~~<h3>Child Component</h3>  
  
<label>Child Component</label>  
<input type="text" />  
  
<p>Value of parent component is: {{ PData }}</p>~~

~~Now, if you type the parent text box, then its value print in the child component. Thus, all is done through the parent to child node via input property.~~

## ~~Step 6: Pass value from child to parent component.~~

~~Passing the data from the child component to the parent component is a little bit tricky. In this scenario, the child component does not have any reference to the parent component.~~

~~So, in this case, we need to emit an event from the child component, and the parent component will listen to it and receive the data via event and display it.~~

~~First, create a reference to the Input in the child component and attach an event listener to it.~~

~~<h3>Child Component</h3>  
  
<label>Child Component</label>  
<input type="text" #ccomponent (keyup)="onChange(ccomponent.value)"/>  
  
<p>Value of parent component is: {{ PData }}</p>~~

~~Write the onChange function in the~~**~~child.component.ts~~**~~file.~~

*~~// child.component.ts~~***~~import~~** ~~{ Component, OnInit, Input, Output, EventEmitter }~~ **~~from~~** ~~'@angular/core';~~**~~@Component~~**~~({  
 selector: 'app-child',  
 templateUrl: './child.component.html',  
 styleUrls: ['./child.component.css']  
})~~**~~export~~****~~class~~** ~~ChildComponent~~ **~~implements~~** ~~OnInit {~~ **~~@Input~~**~~() PData: string = “0”;~~ **~~@Output~~**~~() childEvent =~~ **~~new~~** ~~EventEmitter();~~ **~~constructor~~**~~() { }  
 onChange(value) {~~ **~~this~~**~~.childEvent.emit(value);  
 }  
  
 ngOnInit() {  
 }  
  
}~~

~~When the user types anything in the textbox of the child component, it will start emitting the value from the child component. So, we just need to listen to that event emitter and display the passed value in the parent component.~~

~~Use an event binding in the~~**~~parent.component.html~~**~~file and listen for the event emitter.~~

~~<~~**~~app~~**~~-child [PData]="pcomponent.value" (childEvent)="CData=$event"></~~**~~app~~**~~-child>~~

~~We need to define CData into the~~**~~parent.component.ts~~**~~file.~~

*~~// parent.component.ts~~***~~public~~** ~~CData: number;~~

~~Finally, by interpolation, we can display its value in the~~**~~parent.component.html~~**~~file.~~

~~// parent.component.html  
  
<h3>Parent Component</h3>  
  
<label>Parent Component</label>  
<input type="text" #pcomponent (keyup)="0"/>  
  
<p>Value of child component is: {{ CData }}</p>  
<app-child [PData]="pcomponent.value" (childEvent)="CData=$event"></app-child>~~

## ~~Full Code~~

~~I am writing the following four files if you find any confusion throughout this tutorial.~~

*~~// parent.component.ts~~***~~import~~** ~~{ Component, OnInit }~~ **~~from~~** ~~'@angular/core';~~**~~@Component~~**~~({  
 selector: 'app-parent',  
 templateUrl: './parent.component.html',  
 styleUrls: ['./parent.component.css']  
})~~**~~export~~****~~class~~** ~~ParentComponent~~ **~~implements~~** ~~OnInit {~~ **~~public~~** ~~CData: string = “0”;~~ **~~constructor~~**~~() { }  
  
 ngOnInit() {  
 }  
  
}~~*~~// child.component.ts~~***~~import~~** ~~{ Component, OnInit, Input, Output, EventEmitter }~~ **~~from~~** ~~'@angular/core';~~**~~@Component~~**~~({  
 selector: 'app-child',  
 templateUrl: './child.component.html',  
 styleUrls: ['./child.component.css']  
})~~**~~export~~****~~class~~** ~~ChildComponent~~ **~~implements~~** ~~OnInit {~~ **~~@Input~~**~~() PData: string = “0”;~~ **~~@Output~~**~~() childEvent =~~ **~~new~~** ~~EventEmitter();~~ **~~constructor~~**~~() { }  
 onChange(value: string) {~~ **~~this~~**~~.childEvent.emit(value);  
 }  
  
 ngOnInit() {  
 }  
  
}// child.component.html  
  
<h3>Child Component</h3>  
  
<label>Child Component</label>  
<input type="text" #ccomponent (keyup) = "onChange(ccomponent.value)"/>  
  
<p>Value of parent component is: {{ PData }}</p>~~

~~Finally, our~~**~~Angular Input Output Example~~**~~is over.~~

## Another Example

**Source**: Angular Demos/inout

### Create components

ng g c main

ng g c sub

### main.component.ts

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-main',

templateUrl: './main.component.html',

styleUrls: ['./main.component.css']

})

export class MainComponent implements OnInit {

public childTitle: string = "child title";

public clickCount: number = 0;

constructor() { }

ngOnInit(): void {

}

onCountChanged(count: number): void {

this.clickCount = count;

}

}

### main.component.html

<br>

<h2>main.component</h2>

<input type="text" [(ngModel)]="childTitle">

<h3>Click count: {{ clickCount }}</h3>

<app-sub [title]="childTitle" (onChanged)="onCountChanged($event)"></app-sub>

### sub.component.ts

import { NgTemplateOutlet } from '@angular/common';

import { outputAst } from '@angular/compiler';

import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-sub',

templateUrl: './sub.component.html',

styleUrls: ['./sub.component.css']

})

export class SubComponent implements OnInit {

private \_title: string;

public clickCount: number = 0;

constructor() { }

ngOnInit(): void {

}

@Input()

public set title(title: string) {

this.\_title = title;

}

public get title() {

return this.\_title;

}

@Output()

onChanged = new EventEmitter<number>();

countChange(count: number) {

count++;

this.clickCount = count;

this.onChanged.emit(this.clickCount)

}

}

### sub.component.html

<br>

<h2>sub.component</h2>

<p>Title: {{ title }} </p>

<button (click)="countChange(clickCount)">change count</button>

### Explanation

The “Maincomponent” contain input that is bound to the property “**childTitle**”. This property is referenced in the “Sub component” using the @Input decorator.

We use [(ngModel)] in the “Main component” so that we have two-way binding which lets us change the title of “Sub component” dynamically (*for* ***ngModel*** *we need to add* ***FormsModule*** *in AppModule*).

In Html of the “Sub component”, we created an input attribute [title]. As an input value, we use the “childTitle” property of the “Main component”. That’s how we bind child and parent component using Angular @Input decorator.

The “Sub component” contains a button, and its click event is bound to “**countChange**” method. We also have an EventEmitter “**onChange**” defined using the @Output decorator, which we call when user clicks on the button, which is how the “Maincomponent” will be notified. In EventEmitter, we could pass a parameter, it could be anything, a string or a class etc. In our case, it will be a count number.

EventEmitter in Angular is a type of subscription. EventEmitter helps us to notify other component about some changes.

To have the emitter connect with the “Main component”, we need to change the “Main component” a bit. We need to create method which will be notified and bound to the event emitter in the “Sub component”. For that purpose, we create the method “**onCountChanged**” with a parameter and in the HTML of the “Main component”, we wrote bind the “main” and “sub” components with the following code:

<app-sub [title]="childTitle" (onChanged)="onCountChanged($event)"></app-sub>

That’s it!

# Angular – Directives

<https://www.tutorialspoint.com/angular8/angular8_directives.htm>

Angular directives are DOM elements to interact with your application. Generally, directive is a **TypeScript** function. When this function executes **Angular** compiler checked it inside DOM element. Angular directives begin with **ng-** where **ng** stands for Angular and extends HTML tags with **@directive** decorator.

Directives enables logic to be included in the Angular templates. Angular directives can be classified into three categories and they are as follows −

## Attribute directives

Used to add new attributes for the existing HTML elements to change its look and behaviour.

<HTMLTag [attrDirective]='value' />

For example,

<p [showToolTip]='Tips' />

Here, **showToolTip** refers an example directive, which when used in a HTML element will show tips while user hovers the HTML element.

## Structural directives

Used to add or remove DOM elements in the current HTML document.

<HTMLTag [structuralDirective]='value' />

For example,

<div \*ngIf="isNeeded">

Only render if the \*isNeeded\* value has true value.

</div>

Here, **ngIf** is a built-in directive used to add or remove the HTML element in the current HTML document. Angular provides many built-in directive and we will learn in later chapters.

## Component based directives

Component can be used as directives. Every component has **Input** and **Output** option to pass between component and its parent HTML elements.

<component-selector-name [input-reference]="input-value"> ... </component-selector-name>

For example,

<list-item [items]="fruits"> ... </list-item>

Here, **list-item** is a component and **items** is the input option. We will learn how to create component and advanced usages in the later chapters.

**Example**: **Source**: Angular Demos/directive-app

Create a sample application **(directive-app)**

ng new directive-app

cd directive-app

Create a **test** component using Angular CLI as mentioned below −

ng generate component test

Run the application using below command −

ng serve

## DOM Overview

Let us have a look at DOM model in brief. DOM is used to define a standard for accessing documents. Generally, HTML DOM model is constructed as a tree of objects. It is a standard object model to access html elements.

We can use DOM model in Angular for the below reasons −

* We can easily navigate document structures with DOM elements.
* We can easily add html elements.
* We can easily update elements and its contents.

## Structural directives

**Source**: Angular Demos/directive-app

Structural directives change the structure of **DOM** by adding or removing elements. It is denoted by \* sign with three pre-defined directives **NgIf, NgFor** and **NgSwitch**. Let’s understand one by one in brief.

### NgIf directive

**NgIf** directive is used to display or hide data in your application based on the condition becomes true or false. We can add this to any tag in your template.

Let us try **ngIf** directive in our **directive-app** application.

Add the below tag in **test.component.html**.

<p>test works!</p>

<div \*ngIf="true">Display data</div>

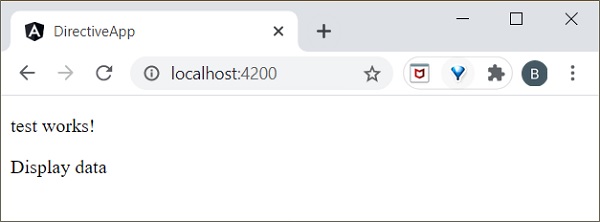
Add the test component in your **app.component.html** file as follows −

<app-test></app-test>

Start your server (if not started already) using the below command −

ng serve

Now, run your application and you could see the below response −



If you set the condition **ngIf=“false”** then, contents will be hidden.

### ngIfElse directive

**ngIfElse** is similar to **ngIf** except, it provides option to render content during failure scenario as well.

Let’s understand how **ngIfElse** works by doing a sample.

Add the following code in **test.component.ts**file.

export class TestComponent implements OnInit {

isLogIn : boolean = false;

isLogOut : boolean = true;

}

Add the following code in **test.component.html** file as follows −

<p>ngIfElse example!</p>

<div \*ngIf="isLogIn; else isLogOut">

Hello you are logged in

</div>

<ng-template #isLogOut>

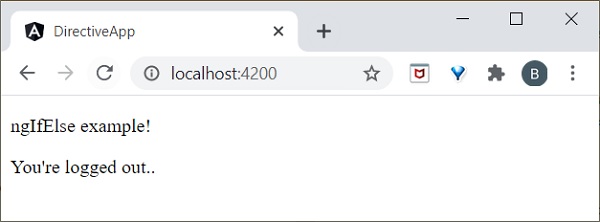
You're logged out..

</ng-template>

Finally, start your application (if not done already) using the below command −

ng serve

Now, run your application and you could see the below response −



Here, isLogOut value is assigned as **true**, so it goes to **else** block and renders **ng-template**. We will learn **ng-template** later.

### ngFor directive

<https://blog.angular-university.io/angular-2-ngfor/>

ngFor is used to repeat a portion of elements from the list of items.

Let’s understand how ngFor works by doing a sample.

Add the list in test.component.ts file as shown below −

list = [1,2,3,4,5];

Add **ngFor** directive in **test.component.html** as shown below −

<h2>ngFor directive</h2>

<ul>

<li \*ngFor="let l of list">

{{l}}

</li>

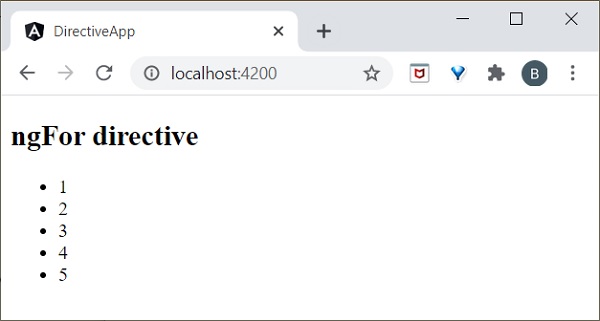
</ul>

Here, the let keyword creates a local variable and it can be referenced anywhere in your template. The let l creates a template local variable to get the list elements.

Finally, start your application (if not done already) using the below command −

ng serve

Now, run your application and you could see the below response −



**Another example**:

Add the following to test.component.ts:

studentArr: any[] = [

{

"id": 1,

"name": "student1"

},

{

"id": 2,

"name": "student2"

},

{

"id": 3, "name": "student3"

},

{

"id": 4,

"name": "student4"

}

];

Add the following to test.component.html:

<div>

   <p>Using ngFor with index:</p>

   <ul>

      <li \*ngFor="let student of studentArr; let i = index">

         {{ i }} : {{student.name}}

      </li>

   </ul>

</div>

### ngFor odd, even, first, last

ng g class classes/Character

ng g c components/ngfordemo

*// Character.ts*

**export** **default** **class** **Character** {

actor\_name: String;

character\_name: String;

gender: String;

status: String;

}

Add following code in ngfordemo.component.ts:

import { Component, OnInit } from '@angular/core';

import { Character } from 'src/app/classes/character';

@Component({

  selector: 'app-ngfordemo',

  templateUrl: './ngfordemo.component.html',

  styleUrls: ['./ngfordemo.component.css']

})

export class NgfordemoComponent {

characters: Character[] = [

{

actor\_name: 'Peter Dinklage',

character\_name: 'Tyrion Lannister',

gender: 'Male',

status: 'Alive'

},

{

actor\_name: 'Sean Bean',

character\_name: 'Ned Stark',

gender: 'Male',

status: 'Dead'

},

{

actor\_name: 'Emilia Clark',

character\_name: 'Khaleesi',

gender: 'Female',

status: 'Alive'

},

{

actor\_name: 'Catelyn Stark',

character\_name: 'Michelle Fairley',

gender: 'Female',

status: 'Dead'

}

];

}

*<!-- ngfordemo.component.html -->*

<div class="container">

<table class="table table-striped">

<thead>

<tr>

<th>Actor Name</th>

<th>Character Name</th>

<th>Gender</th>

<th>Status</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let character of characters">

<td>{{ character.actor\_name }}</td>

<td>{{ character.character\_name }}</td>

<td>{{ character.gender }}</td>

<td>{{ character.status }}</td>

</tr>

</tbody>

</table>

</div>

Add <app-ngfordemo></app-ngfordemo> to app.component.html and view the result.

Let’s add an “index” to the table. Change the ngfordemo.component.html to the following and view the result.

*<!-- ngfordemo.component.html -->*

<div class="container">

<table class="table table-striped">

<thead>

<tr>

<th>Index</th>

<th>Actor Name</th>

<th>Character Name</th>

<th>Gender</th>

<th>Status</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let character of characters; let i = index">

<td>{{ i+1 }}</td>

<td>{{ character.actor\_name }}</td>

<td>{{ character.character\_name }}</td>

<td>{{ character.gender }}</td>

<td>{{ character.status }}</td>

</tr>

</tbody>

</table>

</div>

**Stripe table using even, odd**

**NgForOf** provides exported values that can be aliased to local variables.

The following exported values can be aliased to local variables.

1. $implicit: T: The value of all the individual items in the iterable ([ngForOf](https://angular.io/api/common/NgForOf)).
2. [ngForOf](https://angular.io/api/common/NgForOf): [NgIterable](https://angular.io/api/core/NgIterable)<T>: The value of an iterable expression. Useful when the expression is more complicated then property access, for example when using the async pipe (userStreams | async).
3. index: number: An index of the current item in the iterable.
4. first: boolean: True when an item is the first item in the iterable.
5. last: boolean: True when an item is the last item in the iterable.
6. [even](https://angular.io/api/common/NgForOfContext#even): boolean: True when an item has an even index in the iterable.
7. [odd](https://angular.io/api/common/NgForOfContext#odd): boolean: True when an item has an odd index in the iterable.

Now, we use **even**and **odd**local variables to differentiate the rows of the table.

We use the [**NgClass**](https://appdividend.com/2019/02/07/angular-7-ngclass-tutorial-with-example-angular-ngclass/)conditional directive. Write the following code inside the **ngfordemo.component.css**file. We will define two classes 1) even 2) odd. Then we use those classes based on the **even**and **odd**local variable inside the **ngFor** directive.

.odd {

background-color: beige;

}

.even {

background-color: aquamarine;

}

Change the code in ngfordemo.component.html and view the result:

*<!-- ngfordemo.component.html -->*

<div class="container">

<table class="table table-responsive">

<thead>

<tr>

<th>Index</th>

<th>Actor Name</th>

<th>Character Name</th>

<th>Gender</th>

<th>Status</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let character of characters;

let i = index;

let even = even;

let odd = odd"

[ngClass]="{ odd: odd, even: even }">

<td>{{ i }}</td>

<td>{{ character.actor\_name }}</td>

<td>{{ character.character\_name }}</td>

<td>{{ character.gender }}</td>

<td>{{ character.status }}</td>

</tr>

</tbody>

</table>

</div>

In the above code, see the **ngFor**directive, we have used the local variables like **odd**and **even.**Then we have used the **ngClass**directive to assign the different classes based on the **odd**and **even values.**

**Identifying the first and the last row of a table:**

We have seen the **odd**and **even.**Now, see the **first**and **last element.**

First, define the two classes inside the **ngfordemo.component.css**file.

.first {

background-color: blanchedalmond;

}

.last {

background-color: cadetblue;

}

Now, write the following code inside the **ngfordemo.component.html**file and see the result.

*<!-- ngfordemo.component.html -->*

<div class="container">

<table class="table table-responsive">

<thead>

<tr>

<th>Index</th>

<th>Actor Name</th>

<th>Character Name</th>

<th>Gender</th>

<th>Status</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let character of characters;

let i = index;

let first = first;

let last = last"

[ngClass]="{ first: first, last: last }">

<td>{{ i }}</td>

<td>{{ character.actor\_name }}</td>

<td>{{ character.character\_name }}</td>

<td>{{ character.gender }}</td>

<td>{{ character.status }}</td>

</tr>

</tbody>

</table>

</div>

### ~~trackBy~~

~~Sometimes,~~**~~ngFor~~**~~performance is low with large lists. For example, when adding new item or remove any item in the list may trigger several DOM manipulations. To iterate over large objects collection, we use~~**~~trackBy~~**~~.~~

~~It is used to track when elements are added or removed. It is performed by trackBy method. It has two arguments index and element. Index is used to identity each element uniquely. Simple example is defined below.~~

~~Let’s understand how trackBy works along with~~**~~ngFor~~**~~by doing a sample.~~

~~Add the below code in~~**~~test.component.ts~~**~~file.~~

~~export class TestComponent {~~

~~studentArr: any[] = [ {~~

~~"id": 1,~~

~~"name": "student1"~~

~~},~~

~~{~~

~~"id": 2,~~

~~"name": "student2"~~

~~},~~

~~{~~

~~"id": 3, "name": "student3"~~

~~},~~

~~{~~

~~"id": 4,~~

~~"name": "student4"~~

~~}~~

~~];~~

~~trackByData(index:number, studentArr:any): number {~~

~~return studentArr.id;~~

~~}~~

~~Here, we have created, trackByData() method to access each student element in a unique way based on the id.~~

~~Add the below code in~~**~~test.component.html~~**~~file to define trackBy method inside ngFor.~~

~~<ul>~~

~~<li \*ngFor="let std of studentArr; trackBy: trackByData">~~

~~{{std.name}}~~

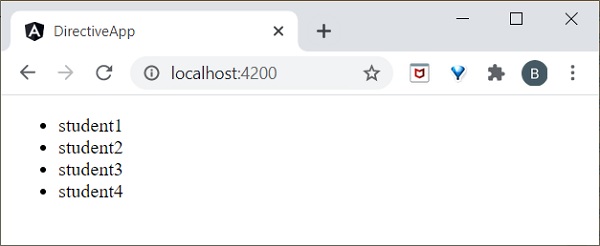
~~</li>~~

~~</ul>~~

~~Finally, start your application (if not done already) using the below command −~~

~~ng serve~~

~~Now, run your application and you could see the below response −~~

~~~~

~~Here, the application will print the student names. Now, the application is tracking student objects using the student id instead of object references. So, DOM elements are not affected.~~

### NgSwitch directive

**NgSWitch** is used to check multiple conditions and keep the DOM structure as simple and easy to understand.

Let us try **ngSwitch** directive in our **directive-app** application.

Add the following code in **test.component.ts** file.

export class TestComponent implements OnInit {

logInName = 'admin';

}

Add the following code in test.component.html file as follows −

<h2>ngSwitch directive</h2>

<ul [ngSwitch]="logInName">

<li \*ngSwitchCase="'user'">

<p>User is logged in..</p>

</li>

<li \*ngSwitchCase="'admin'">

<p>admin is logged in</p>

</li>

<li \*ngSwitchDefault>

<p>Please choose login name</p>

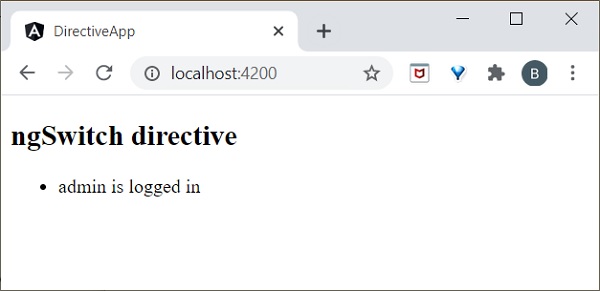
</li>

</ul>

Finally, start your application (if not done already) using the below command −

ng serve

Now, run your application and you could see the below response −



Here, we have defined **logInName** as **admin.** So, it matches second SwitchCase and prints above admin related message.

## Attribute directives

**Source**: Angular Demos/directive-app

Attribute directives performs the appearance or behavior of DOM elements or components. Some of the examples are NgStyle, NgClass and NgModel. Whereas, NgModel is two-way attribute data binding explained earlier.

### ngStyle

**ngStyle** directive is used to add dynamic styles. Below example is used to apply blue color to the paragraph.

Let us try **ngStyle** directive in our **directive-app** application.

Add below code in test.component.ts file:

export class AttrDirectivesComponent implements OnInit {

  serverID: number = 10;

  serverStatus: string = 'Offline';

  constructor() {

    this.serverStatus = Math.random() > 0.5 ? 'Online' : 'Offline';

  }

  getServerStatus() {

    return this.serverStatus;

  }

  getColor() {

    return this.serverStatus === 'Online' ? 'green' : 'red';

  }

}

Add below content in **test.component.html** file.

<!-- ngStyle -->

<p [ngStyle]="{'color': 'blue', 'font-size': '14px'}">

    paragraph style is applied using ngStyle

</p>

<div>

    <p [ngStyle]="{backgroundColor: getColor()}"

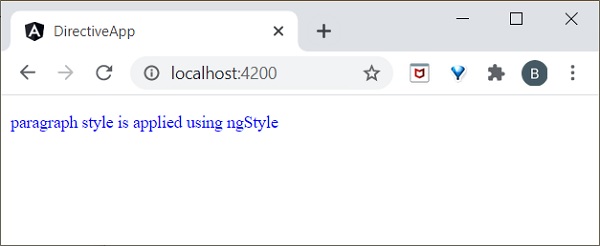
    [ngClass]="{Online: serverStatus === 'Online'}">  Server  with ID {{serverID}} is {{serverStatus}}. </p>

</div>

Start your application (if not done already) using the below command −

ng serve

Now, run your application and you could see the below response −



### ngClass

**ngClass** is used to add or remove CSS classes in HTML elements.

Let us try **ngClass** directive in our **directive-app** application.

Create a class **User** using the below command

ng g class User

Move to **src/app/user.ts** file and add the below code −

export class User {

userId : number; userName : string;

}

Here, we have created two property **userId** and **userName** in the **User** class.

Open **test.component.ts** file and add the below changes −

import { User } from '../user';

export class TestComponent implements OnInit {

users: User[] = [

{

"userId": 1,

"userName": 'User1'

},

{

"userId": 2,

"userName": 'User2'

},

];

}

Here, we have declared a local variable, users and initialise with 2 users object.

Open **test.component.css** file and add below code

.highlight {

color: red;

}

Open your **test.component.html** file and add the below code −

<div class="container">

<br/>

<div \*ngFor="let user of users" [ngClass]="{

'highlight':user.userName === 'User1'

}">

{{ user.userName }}

</div>

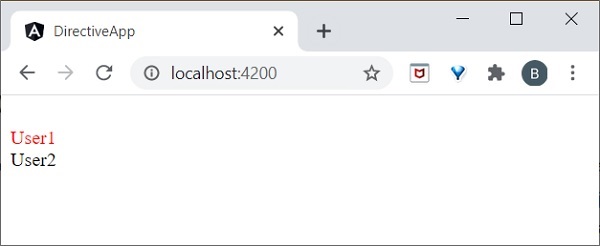
</div>

Here, we have applied, **ngClass** for **User1** so it will highlight the **User1**.

Finally, start your application (if not done already) using the below command −

ng serve

Now, run your application and you could see the below response −



## Custom directives

**Source**: Angular Demos/directive-app

Angular provides option to extend the angular directive with user defined directives and it is called **Custom directives**. Let us learn how to create custom directive in this chapter.

Let us try to create custom directive in our **directive-app** application.

Angular CLI provides a below command to create custom directive.

ng generate directive customstyle

Open **app.module.ts.** The directive will be configured in the **AppModule** through **declarations** meta data.

import { CustomstyleDirective } from './customstyle.directive';

@NgModule({

declarations: [

AppComponent,

TestComponent,

CustomstyleDirective

]

})

Open **customstyle.directive.ts** file and add the below code −

import { Directive, ElementRef } from '@angular/core';

@Directive({

selector: '[appCustomstyle]'

})

export class CustomstyleDirective {

constructor(el: ElementRef) {

el.nativeElement.style.fontSize = '24px';

}

}

Here, **constructor** method gets the element using **CustomStyleDirective** as **el**. Then, it accesses el’s style and set its font size as **24px** using CSS property.

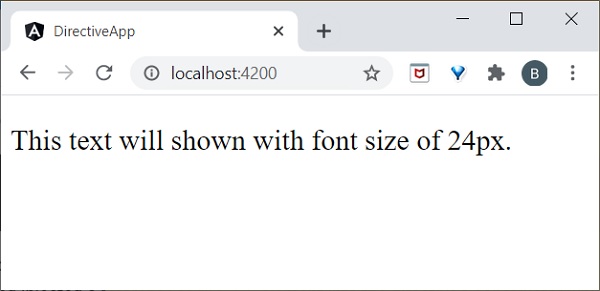
Open test.component.html and add the following:

<h1 appCustomstyle>This text is formatted using a custom style directive.</h1>

Finally, start your application (if not done already) using the below command −

ng serve

Now, run your application and you could see the below response −



### ng-template

**ng-template** is used to create dynamic and reusable templates. It is a virtual element. If you compile your code with **ng-template** then is converted as comment in DOM.

For example,

Let’s add a below code in **test.component.html** page.

<h3>ng-template</h3>

<ng-template>ng-template tag is a virtual element</ng-template>

If you run the application, then it will print only **h3** element. Check your page source, template is displayed in comment section because it is a virtual element so it does not render anything. We need to use **ng-template** along with Angular directives.

Normally, directive emits the HTML tag it is associated. Sometimes, we don’t want the tag but only the content. For example, in the below example, li will be emitted.

<li \*ngFor="let item in list">{{ item }}</li>

We can use **ng-template** to safely skip the **li** tag.

### ng-template with structural directive

**ng-template** should always be used inside **ngIf, ngFor** or **ngSwitch** directives to render the result.

Let’s assume simple code.

<ng-template [ngIf]=true>

<div><h2>ng-template works!</h2></div>

</ng-template>

Here, if **ngIf** condition becomes true, it will print the data inside div element. Similarly, you can use **ngFor** and **ngSwitch** directives as well.

### NgForOf directive

**ngForOf** is also a structural directive used to render an item in a collection. Below example is used to show **ngForOf** directive inside **ng-template**.

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-test',

template: `

<div>

<ng-template ngFor let-item [ngForOf]="Fruits" let-i="index">

<p>{{i}}</p>

</ng-template>

</div>`

,

styleUrls: ['./test.component.css']

})

export class TestComponent implements OnInit {

Fruits = ["mango","apple","orange","grapes"];

ngOnInit()

{

}

}

If you run the application, it will show the index of each elements as shown below −

0

1

2

3

#### ngFor vs ngForOf

* [ngFor] is ***not*** **type safe**
* [NgForOf] is **type safe**

Because both class details are little different

* ngFor Class type is any type
* But ngForOf class type is generic ngForOf : NgIterable<T>

## ngFor and ngForOf Example

<https://stackblitz.com/edit/angular-ngfor-ngforof?file=src%2Fapp%2Fapp.component.html>

**Employee.ts**

export class Employee {

  constructor(public id:number,public name:string,public salary:number){}

}

**app.component.ts:**

import { Component } from '@angular/core';

import {Employee} from './Employee';

@Component({

  selector: 'my-app',

  templateUrl: './app.component.html',

  styleUrls: [ './app.component.css' ]

})

export class AppComponent  {

  employees:Employee[]=[];

  constructor(){

    let emp1 = new Employee(100,"Raj",12000);

    let emp2 = new Employee(200,"Ajay",18000);

    let emp3 = new Employee(300,"Seeta",14000);

    let emp4 = new Employee(400,"Reeta",16000);

    this.employees.push(emp1);

    this.employees.push(emp2);

    this.employees.push(emp3);

    this.employees.push(emp4);

  }

  fun(index,data){

    console.log(index+"--"+data.id)

  }

}

**app.component.html:**

<div>

  <h2>Using \*ngFor loop</h2>

  <table border="1">

    <tr>

      <th>Index</th>

      <th>Id</th>

      <th>Name</th>

      <th>Salary</th>

    </tr>

    <tr \*ngFor="let emp of employees;let i =index;trackBy:fun" >

          <td>{{i+1}}</td>

          <td>{{emp.id}}</td>

          <td>{{emp.name}}</td>

          <td>{{emp.salary}}</td>

    </tr>

  </table>

  <h2>Using \*ngForOf loop</h2>

  <table border="1">

    <tr>

      <th>Index</th>

      <th>Id</th>

      <th>Name</th>

      <th>Salary</th>

    </tr>

    <ng-template ngFor let-emp1="$implicit" [ngForOf]="employees" let-i="index">

      <tr>

        <td>{{i+1}}</td>

        <td>{{emp1.id}}</td>

        <td>{{emp1.name}}</td>

        <td>{{emp1.salary}}</td>

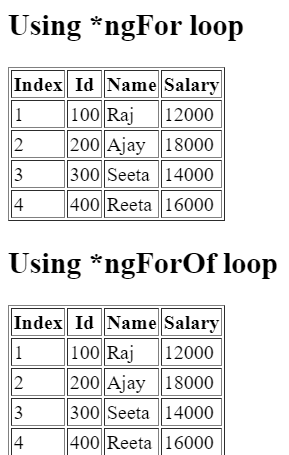
      </tr>

    </ng-template>

  </table>

</div>

**Result:**



## Creating a Custom Directive

<https://codecraft.tv/courses/angular/custom-directives/creating-a-custom-directive/>

**Source**: Angular Demos/directive-app

### [Directive Decorator](https://codecraft.tv/courses/angular/custom-directives/creating-a-custom-directive/#_directive_decorator)

We’ll call our directive ccCardHover and we’ll attach it to the card block like so:

Copy<div class="card card-block" ccCardHover>...</div>

**Tip**

The Angular team recommends using directives as attributes, prefixed with a namespace. We’ve prefixed our directive with the namespace 'cc'.

We create directives by annotating a class with the @Directive decorator.

Let’s create a class called CardHoverDirective and use the @Directive dectorator to associate this class with our attribute ccCardHover, like so:

ng g directive directives/CardHover

import { Directive } from '@angular/core';

@Directive({

  selector: '[appCardHover]'

})

export class CardHoverDirective {

  constructor() { }

}

### [Attribute Selector](https://codecraft.tv/courses/angular/custom-directives/creating-a-custom-directive/#_attribute_selector)

The above code is very similar to what we would write if this was a component, the first striking difference is that the selector is wrapped with `[]`.

To understand why we do this we first need to understand that the selector attribute uses CSS matching rules to match a component/directive to a HTML element.

In CSS to match to a specific element we would just type in the name of the element, so input {…​}`or `p {…​}.

This is why previously when we defined the selector in the @Component directive we just wrote the name of the element, which matches onto an element of the same name.

If we wrote the selector as .ccCardHover, like so:

import { Directive } from '@angular/core';

@Directive({

  selector: “.ccCardHover”

})

export class CardHoverDirective {

  constructor() { }

}

Then this would associate the directive with any element that has a *class* of ccCardHover, like so:

<div class="card card-block ccCardHover">...</div>

We want to associate the directive to an element which has a certain attribute.

To do that in CSS we wrap the name of the attribute with [], and this is why the selector is called [ccCardHover].

### [Directive Constructor](https://codecraft.tv/courses/angular/custom-directives/creating-a-custom-directive/#_directive_constructor)

The next thing we do is add a constructor to our directive, like so:

import { ElementRef } from '@angular/core';

.

.

.

class CardHoverDirective {

constructor(private el: ElementRef) {

}

}

When the directive gets created Angular can inject an instance of something called ElementRef into its constructor.

**Note**

How this works is called Dependency Injection, it’s a really important aspect of Angular and we discuss this in detail in a later section.

The ElementRef gives the directive direct access to the DOM element upon which it’s attached.

Let’s use it to change the background color of our card to gray.

ElementRef itself is a wrapper for the actual DOM element which we can access via the property nativeElement, like so:

el.nativeElement.style.backgroundColor = "gray";

.

.

class CardHoverDirective {

constructor(private el: ElementRef)

el.nativeElement.style.backgroundColor = "gray";

}

}

### Full Code

**CardHoverDirective:**

ng g directive directives/CardHover

import { Directive, ElementRef, Input } from '@angular/core';

@Directive({

  selector: '[ccCardHover]'

})

export class CardHoverDirective {

  constructor(private el: ElementRef) {

    // this.el.nativeElement.classList.add(this.ttClass)

    this.el.nativeElement.style.backgroundColor = "gray"

    }

}

**Joke Entity Class:**

ng g class classes/joke

export class Joke {

    public setup: string;

    public punchline: string;

    public hide: boolean;

    constructor(setup: string, punchline: string) {

      this.setup = setup;

      this.punchline = punchline;

      this.hide = true;

    }

    toggle() {

      this.hide = !this.hide;

    }

}

**Joke Component:**

ng g class components/joke

import { Component, Input, OnInit } from '@angular/core';

import { Joke } from 'src/app/classes/joke';

@Component({

  selector: 'joke',

  templateUrl: './joke.component.html',

  styleUrls: ['./joke.component.css']

})

export class JokeComponent implements OnInit {

  @Input('joke') data!: Joke;

  constructor() { }

  ngOnInit(): void {

  }

}

**Joke List Component:**

ng g class components/JokeList

import { Component, OnInit } from '@angular/core';

import { Joke } from 'src/app/classes/joke';

@Component({

  selector: 'joke-list',

  templateUrl: './joke-list.component.html',

  styleUrls: ['./joke-list.component.css']

})

export class JokeListComponent implements OnInit {

  jokes!: Joke[];

  constructor() {

    this.jokes = [

      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me (Halloumi)"),

      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-pony (Mascarpone)"),

      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very mature’"),

    ];

  }

  ngOnInit(): void {

  }

}

**joke.component.html:**

<div class="card card-block" ccCardHover>

    <h4 class="card-title">{{data.setup}}</h4>

    <p class="card-text"

       [hidden]="data.hide">{{data.punchline}}</p>

    <button (click)="data.toggle()"

       class="btn btn-primary">Tell Me

    </button>

</div>

**joke-list.component.html:**

<joke \*ngFor="let j of jokes" [joke]="j"></joke>

**app.component.css:**

.blue {

  background-color: lightblue;

}

**app.component.html:**

<joke-list></joke-list>

## Component directives

Component directives are based on component. Actually, each component can be used as directive. Component provides @Input and @Output decorator to send and receive information between parent and child components. We have see an example earlier for @Input and @Output.

## Sample Working example

Let us add a new component in our **ExpenseManager** application to list the expense entries.

**Source**: Angular Demos/directive-app

Create a new component, **ExpenseEntryListComponent** using below command −

ng generate component components/ExpenseEntryList

Here, the command creates the ExpenseEntryList Component and update the necessary code in **AppModule**.

Create a class:

ng g class classes/ExpenseEntry

Modify the class with the following code:

export class ExpenseEntry {

    id: number = 0;

    item: string = "";

    amount: number = 0;

    category: string = "";

    location: string = "";

    spendOn: Date = new Date();

    createdOn: Date = new Date();

}

Import **ExpenseEntry** into **ExpenseEntryListComponent** component **(**expense-entry-list.component.ts**):**

import { ExpenseEntry } from 'src/app/classes/expense-entry';

Add a method, getExpenseEntries() to return list of expense entry (mock items) in ExpenseEntryListComponent:

getExpenseEntries() : ExpenseEntry[] {

let mockExpenseEntries : ExpenseEntry[] = [

{ id: 1,

item: "Pizza",

amount: Math.floor((Math.random() \* 10) + 1),

category: "Food",

location: "Mcdonald",

spendOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10),

createdOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10) },

{ id: 1,

item: "Pizza",

amount: Math.floor((Math.random() \* 10) + 1),

category: "Food",

location: "KFC",

spendOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10),

createdOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10) },

{ id: 1,

item: "Pizza",

amount: Math.floor((Math.random() \* 10) + 1),

category: "Food",

location: "Mcdonald",

spendOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10),

createdOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10) },

{ id: 1,

item: "Pizza",

amount: Math.floor((Math.random() \* 10) + 1),

category: "Food",

location: "KFC",

spendOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10),

createdOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10) },

{ id: 1,

item: "Pizza",

amount: Math.floor((Math.random() \* 10) + 1),

category: "Food",

location: "KFC",

spendOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10),

createdOn: new Date(2020, 4, Math.floor((Math.random() \* 30) + 1), 10, 10, 10)

},

];

return mockExpenseEntries;

}

Declare a local variable, expenseEntries in expense-entry-list.component.ts and load the mock list of expense entries as mentioned below −

title: string;

expenseEntries: ExpenseEntry[];

constructor() { }

ngOnInit() {

this.title = "Expense Entry List";

this.expenseEntries = this.getExpenseEntries();

}

Open the template file (expense-entry-list.component.html) and show the mock entries in a table.

<!-- Page Content -->

<div class="container">

<div class="row">

<div class="col-lg-12 text-center" style="padding-top: 20px;">

<div class="container" style="padding-left: 0px; padding-right: 0px;">

<div class="row">

<div class="col-sm" style="text-align: left;">

{{ title }}

</div>

<div class="col-sm" style="text-align: right;">

<button type="button" class="btn btn-primary">Edit</button>

</div>

</div>

</div>

<div class="container box" style="margin-top: 10px;">

<table class="table table-striped">

<thead>

<tr>

<th>Item</th>

<th>Amount</th>

<th>Category</th>

<th>Location</th>

<th>Spent On</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let entry of expenseEntries">

<th scope="row">{{ entry.item }}</th>

<th>{{ entry.amount }}</th>

<td>{{ entry.category }}</td>

<td>{{ entry.location }}</td>

<td>{{ entry.spendOn | date: 'short' }}</td>

</tr>

</tbody>

</table>

</div>

</div>

</div>

</div>

Here,

* Used bootstrap table. **table** and **table-striped** will style the table according to Boostrap style standard.
* Used **ngFor** to loop over the **expenseEntries** and generate table rows.

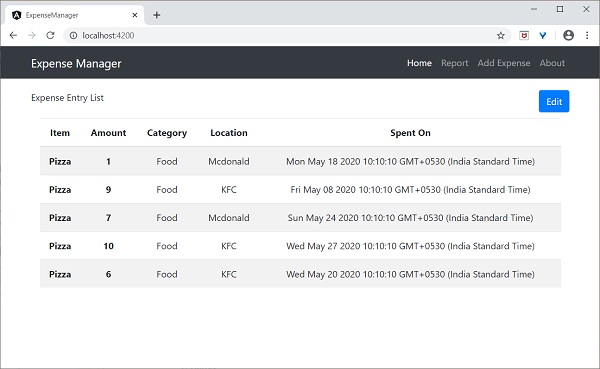
**REF**: https://angular.io/api/common/DatePipe

Open AppComponent template, src/app/app.component.html and include ExpenseEntryListComponent as shown below −

...

<app-expense-entry-list></app-expense-entry-list>

Finally, the output of the application is as shown below.



# Bind Select Dropdown List

<https://www.talkingdotnet.com/bind-select-dropdown-list-in-angular-8/>

**Class product.ts:**

export class Product {

    id: number = 0;

    name: string = "";

}

**dropdown.component.ts**

import { Component, OnInit } from '@angular/core';

import { Product } from 'src/app/classes/product';

@Component({

  selector: 'app-dropdown',

  templateUrl: './dropdown.component.html',

  styleUrls: ['./dropdown.component.css']

})

export class DropdownComponent implements OnInit {

  public products: Product[] = [

    { id: 1, name: "Table" },

    { id: 2, name: "Chair" },

    { id: 3, name: "Light" }

  ];

  public selectedProduct: Product = this.products[0];

  onSelect(event: Event) {

    // this.selectedProduct = null;

    let evt = event.target as HTMLInputElement;

    let productId = Number(evt.value);

    for (var i = 0; i < this.products.length; i++) {

      if (this.products[i].id == productId) {

        this.selectedProduct = this.products[i];

      }

    }

  }

  constructor() { }

  ngOnInit(): void {

  }

}

dropdown.component.html

<h2>Bind to Dropdown</h2>

<select (change)="onSelect($event)">

    <option \*ngFor="let product of products" [value]="product.id">{{product.name}}</option>

  </select>

<div>selection={{selectedProduct.name}}</div>

**app.component.html:**

<app-dropdown></app-dropdown>

# Angular – Dependency Injection (DI) and Services

<https://www.freecodecamp.org/news/angular-dependency-injection/>

**Source**: Angular Demos/di-app

## **Motivation**

Dependency Injection is often more simply referred to as DI. The paradigm exists throughout Angular. It keeps code flexible, testable, and mutable. Classes can inherit external logic without knowing how to create it. Any consumers of those classes also do not need to know anything.

DI saves classes and consumers alike from having to know more than necessary. Yet the code is as modular as it was before thanks to the mechanisms supporting DI in Angular.

Services are a key benefactor of DI. They rely on the paradigm for injection into various consumers. Those consumers can then take advantage of that service provides and/or forward it elsewhere.

Services are not alone. Directives, pipes, components, and so on: every schematic in Angular benefits from DI in some way or another.

## Injectors

Injectors are data structures that store instructions detailing where and how services form. They act as intermediaries within the Angular DI system.

Module, directive, and component classes contain metadata specific to injectors. A new injector instance accompanies every one of these classes. In this way, the application tree mirrors its hierarchy of injectors.

The providers: [] metadata accepts services that then register with the class’ injector. This provider field adds the instructions necessary for an injector to function. A class (assuming it has dependencies) instantiates a service by taking on its class as its data type. The injector aligns this type a creates an instance of that service on the class’ behalf.

Of course, the class can only instantiate what the injector has instructions for. If the class’ own injector does not have the service registered, then it queries its parent. So on and so forth until either reaching an injector with the service or the application root.

Services can register at any injector within the application. Services go in the providers: [] metadata field of class modules, directives, or components. The class’ children can instantiate a service registered in the class’ injector. Child injectors fallback on parent injectors after all.

## Dependency Injection

Take a look at the skeletons for each class: service, module, directive, and component.

// service

import { Injectable } from '@angular/core';

@Injectable({

providedIn: /\* injector goes here \*/

})

export class TemplateService {

constructor() { }

}

// module

import { NgModule } from '@angular/core';

import { CommonModule } from '@angular/common';

@NgModule({

imports: [

CommonModule

],

declarations: [],

providers: [ /\* services go here \*/ ]

})

export class TemplateModule { }

// directive

import { Directive } from '@angular/core';

@Directive({

selector: '[appTemplate]',

providers: [ /\* services go here \*/ ]

})

export class TemplateDirective {

constructor() { }

}

//component

import { Component } from '@angular/core';

@Component({

selector: 'app-template',

templateUrl: './template.component.html',

styleUrls: ['./template.component.css'],

providers: [ /\* services go here \*/ ]

})

export class TemplateComponent {

// class logic ...

}

Each skeleton can register services to an injector. In fact, TemplateService is a service. As of Angular 6, services can now register with injectors using @Injectable metadata.

### **In Any Case**

Notice the providedIn: string (@Injectable) and providers: [] (@Directive, @Componet and @Module) metadata. They tell injectors where and how to create a service. Otherwise, injectors would not know how to instantiate.

What if a service has dependencies? Where would the results go? Providers answers those question so that injectors can instantiate properly.

Injectors form the backbone of the DI framework. They store instructions to instantiate services so consumers do not have to. They receive service instances without needing to know anything about the source dependency!

I should also note that other schematics without injectors can still utilize dependency injection. They cannot register additional services, but they can still instantiate from injectors.

## Service

The providedIn: string metadata of @Injectable specifies which injector to register with. Using this method, and depending on if the service gets used, the service may or may not register with the injector. Angular calls this tree-shaking.

By default the value is set to ‘root’. This translates to the root injector of the application. Basically, setting the field to ‘root’ makes the service available anywhere.

**Quick Note**

As previously mentioned, child injectors fallback on their parents. This fallback strategy ensures parents do not have to re-register for every injector. Refer to this article on [Services and Injectors](https://guide.freecodecamp.org/angular/services-and-injectors) for an illustration of this concept.

Registered services are singletons. Meaning, the instructions to instantiate the service exists on only one injector. This assumes it has not been explicitly registered elsewhere.

## Module, Directive, and Component

Modules and components each have their own injector instance. This is evident given the providers: [] metadata field. This field takes an array of services and registers them with the injector of the module or component class. This approach happens in the @NgModule, @Directive, or @Component decorators.

This strategy omits tree-shaking, or the optional removal of unused services from injectors. Service instances live on their injectors for the life of the module or component.

## Instantiating References

References to the DOM can instantiate from any class. Keep in mind that references are still services. They differ from traditional services in representing the state of something else. These services include functions to interact with their reference.

Directives are in constant need of DOM references. Directives perform mutations on their host elements through these references. See the following example. The directive’s injector instantiates a reference of the host element into the class’ constructor.

// directives/highlight.directive.ts

import { Directive, ElementRef, Renderer2, Input } from '@angular/core';

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(

private renderer: Renderer2,

private host: ElementRef

) { }

@Input() set appHighlight (color: string) {

this.renderer.setStyle(this.host.nativeElement, 'background-color', color);

}

}

<!--app.component.html-->

<p [appHighlight]="'yellow'">Highlighted Text!</p>

Renderer2 also gets instantiated. Which injector do these services come from? Well, each service’s source code comes from @angular/core. These services must then register with the application’s root injector.

// app.component.html

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

import { HighlightDirective } from './directives/highlight.directive';

@NgModule({

declarations: [

AppComponent,

HighlightDirective

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [

AppComponent

]

})

export class AppModule { }

An empty providers array!? Not to fear. Angular registers many services with the root injector automatically. This includes ElementRef and Renderer2. In this example, we are managing the host element through its interface stemming from the instantiation of ElementRef. Renderer2 lets us update the DOM through Angular’s view model.

It is important recognize the role that injectors play in the above example. By declaring variable types in the constructor, the class obtains valuable services. Each parameter’s data type maps to a set of instructions within the injector. If the injector has that type, it returns an instance of said type.

## Instantiating Services

The [Services and Injectors](https://guide.freecodecamp.org/angular/services-and-injectors) article explains this section to an extent. Though, this section rehashes the previous section or the most part. Services will often provide references to something else. They may just as well provide an interface extending a class’ capabilities.

The next example will define a logging service that gets added to a component’s injector via its providers: [] metadata.

// directives/highlight.directive.ts

import { Directive, ElementRef, Renderer2, Input } from '@angular/core';

@Directive({

  selector: '[appHighlight]'

})

export class HighlightDirective {

  constructor(

    private renderer: Renderer2,

    private host: ElementRef

  ) { }

  @Input() set appHighlight (color: string) {

    this.renderer.setStyle(this.host.nativeElement, 'background-color', color);

  }

}

// services/logger.service.ts

import { Injectable } from '@angular/core';

@Injectable({

  providedIn: 'root'

})

export class LoggerService {

  callStack: string[] = [];

  addLog(message: string): void {

    this.callStack = [message].concat(this.callStack);

    this.printHead();

  }

  clear(): void {

    this.printLog();

    this.callStack = [];

    console.log("DELETED LOG");

  }

  private printHead(): void {

    console.log("printHead()...")

    console.log(this.callStack[0] || null);

  }

  private printLog(): void {

    console.log("printLog()...")

    this.callStack.reverse().forEach((log) => console.log(log));

  }

}

// app.component.ts

import { Component } from '@angular/core';

import { LoggerService } from './services/logger.service';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'di-app';

  constructor(private logger: LoggerService) {

  }

  logMessage(event: any, message: string): void {

    event.preventDefault();

    this.logger.addLog(`Message: ${message}`);

  }

  clearLog(): void {

    this.logger.clear();

  }

}

<!-- app.component.html -->

<p [appHighlight]="'yellow'">Highlighted Text!</p>

<h1>Log Example</h1>

<form (submit)="logMessage($event, userInput.value)">

  <input #userInput placeholder="Type a message...">

  <button type="submit">SUBMIT</button>

</form>

<h3>Delete Logged Messages</h3>

<button type="button" (click)="clearLog()">CLEAR</button>

Focus on the AppComponent constructor and metadata. The component injector receives instructions from the provider’s metadata field containing LoggerService. The injector then knows what to instantiate LoggerService from requested in the constructor.

The constructor parameter loggerService has the type LoggerService which the injector recognizes. The injector follows through with the instantiation as mentioned.

# Promises in TypeScript

<https://medium.com/jspoint/typescript-promises-and-async-await-b842b55ee3fd>

## Promises

A **promise** is an instance (object) of the [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) class (constructor). To create a promise, we use new Promise(executor) syntax and provide an **executor** function as an argument. This **executor** function provides a means to control the behavior of our promise **resolution** or **rejection**.

In TypeScript, we can provide the **data type of the value returned** when promise fulfills. Since the error returned by the promise can take any shape, the default data type of value returned when the promise is rejected is set to any by the TypeScript.

To annotate the resolution value type of the promise, we use a generic type declaration. Basically, you promise a type with Promise constructor in the form of new Promise<Type>() which indicates the resolved value type of the promise. But you can also use let p: Promise<Type> = new Promise() syntax to achieve the same.

**Promise Syntax**:

*// Promises Syntax:*

const myPromise = new Promise((*resolve*, *reject*) => {

  Do some asynchronous operation

  If the operation is successful, call resolve with the result

  If the operation fails, call reject with an error object

});

myPromise

  .then((*result*) => {

    Handle the successful result

  })

  .catch((*error*) => {

    Handle the error

  });

**Example**: PromiseRandomInt.ts

*// Returns a random int b/w 0 and 9.*

const getRandomInt = (): string => {

    return (Math.random() \* 10).toFixed(0);

};

*// Resolve with an 'even' integer.*

const findEven = new Promise<number>((*resolve*, *reject*) => {

    setTimeout(function (): void {

*// Convert string to number.*

        const value = parseInt(getRandomInt());

        if (value % 2 === 0) {

            resolve(value);

        }

        else {

            reject("Odd number found. " + value);

        }

    }, 1000);

});

*// Listen to the promise resolution.*

findEven.then((*value*) => {

*// (param) value: number.*

    console.log('Resolved:', value);

}).catch((*error*) => {

*// (param) error: any.*

    console.log('Rejected:', error);

}).finally( () => {

    console.log('Completed.');

});

Runt it as:

tsc PromiseRandomInt.ts

node PromiseRandomInt.ts

OR

npx ts-node .\PromiseRandomInt.ts

If the TypeScript compiler complains about the finally method, that means your TypeScript compiler doesn’t import type definitions for the finally method. This method was introduced in **ES2016**, hence it’s quite new. Other features of the Promise API used in this lesson are pretty new, hence make sure your tsconfig.json file has all the new libraries loaded.

*// tsconfig.json*

{

    "files": [

        "./PromiseRandomInt.ts"

    ],

    "compilerOptions": {

        "lib": [

            "DOM",

            "ES2020"

        ]

    }

}

## Promise Chaining

The then, catch and finally methods **return a promise** implicitly. Any value returned by these callback functions is wrapped with a promise and returned, including undefined. This implicit promise is **resolved by default** unless you are deliberately returning a new promise from these methods that could fail.

Hence you can append then, catch or finally methods to any of the previous then, catch or finally method. If an implicit promise is returned by one of these methods, then the resolved value type of this implicit promise is the type of the returned value. Let’s see a quick example.

**Example**: promise-chaining.ts

*// Returns a random int b/w 0 and 9.*

const getRandomIntChain = (): string => {

    return (Math.random() \* 10).toFixed(0);

};

*// Resolve with an 'even' integer.*

const findEvenChain = new Promise<number>((*resolve*, *reject*) => {

    setTimeout(function (): void {

*// Convert string to number.*

        const value = parseInt(getRandomIntChain());

        if (value % 2 === 0) {

            resolve(value);

        }

        else {

            reject("Odd number found. " + value);

        }

    }, 1000);

});

*// Listen to the promise resolution.*

findEvenChain.then((*value*) => {

*// (param) value: number.*

    console.log('Resolved-1:', value + 1);

    return `${value + 1}`;

}).then( (*value*) => {

*// (param) value: string.*

    console.log('Resolved-2:', value + 1);

}).catch((*error*) => {

*// (param) error: any.*

    console.log('Rejected:', error);

}).finally(() => {

    console.log('Completed.');

});

We have modified the previous example and added another then method to the first then method. Since the first then method returns a value of type string, the implicit promise returned by this method will be resolved with a value of the type string. Hence, the second then method will receive value argument of type string as you can see from the results.

## Promise.resolve

The resolve **static** method of the Promise call returns a promise that is already resolved successfully with a value that you provide in the Promise.resolve(value) call. This is easier than creating a new instance of the Promise call and adding logic to resolve the promise immediately.

**Example**: promise-resolve.ts

*// Returns a random int b/w 0 and 9.*

const getRandomIntResolve = (): string => {

    return (Math.random() \* 10).toFixed(0);

};

*// Get random integer of type 'number'.*

const value = parseInt(getRandomIntResolve());

*// const numPromise: Promise<number>.*

const numPromise = Promise.resolve(value);

*// Listen to the promise resolution.*

numPromise.then( (*value*) => {

*// (param) value: number.*

    console.log('Resolved:', value + 1);

}).catch( (*error*) => {

    console.log('Rejected.', error);

});

As you can see from the above results, the promise returned by the Promise.resolve(value) call **always resolves immediately** with a number value since the value argument has the type of number.

## Promise.reject

Similar to Promise.resolve static method, the Promise.reject(error) method always returns a **rejected** promise. The value of the promise rejection is taken from the error argument and its type is any.

**Example**: promise-reject.ts

*// Sample error msg.*

const message = 'Oops! Something went wrong.';

*// const oopsPromise: Promise<never>.*

const oopsPromise = Promise.reject(message);

*// Listen to the promise resolution.*

oopsPromise.then( (*value*) => {

*// (param) value: never.*

    console.log('Resolved.', value);

}).catch((*error*) => {

    console.log('Rejected.', error);

});

The type of promise returned by the Promise.reject method is Promise<never> because this promise never resolves, so there won’t be any promise resolution value. Therefore the type of the value resolved by the promise is never as never signifies the value that never occurs.

## Promise.all

In some scenarios, you are dealing with multiple promises. If you want to execute a callback function when all the promises are resolved successfully, use the Promise.[all](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all) static method.

var pAll = Promise.all([ p1, p2, ... ])pAll.then( ( [ r1, r2, ... ] ) => {\_\_\_});

The Promise.all method takes an array (iterable precisely) of promises and returns a new promise. The returned promise pAll is resolved when all promises p1, p2, ... are resolved successfully. This promise is resolved with an array value that contains the promise resolution values of p1, p2, ... in the order of their appearance.

**Example**: promise-all.ts

*// A function that returns a delayed promise.*

const getPromise = (

*value*: number, *delay*: number, *fail*: boolean

): Promise<number> => {

    return new Promise<number>( (*resolve*, *reject*) => {

        setTimeout(

            () => fail ? reject(value) : resolve(value),

            delay

        );

    });

};

*//const allPromises: Promise<number[]>.*

const allPromises = Promise.all<number>( [

    getPromise(0,0,false),      *// 0s.*

    getPromise(1,2000,false),    *// 2s.*

    getPromise(2,1000,false),    *// 1s.*

*//getPromise(3,1500,true),    // 1.5s. Rejects.*

]);

*// Listen to allPromises resolution.*

console.time('settled in...');

allPromises.then( (*value*) => {

*// (param) value: number[].*

    console.log('Resolved.', value);

}).catch((*error*) => {

*// (param) value: error.*

    console.log('Rejected.', error);

}).finally( () => {

    console.timeEnd('settled in...');

});

As you can see from the above example, Promise.all method is **generic** and it takes the type the value resolved by each promise provided to it. Providing a generic type is quite helpful when we are using the resolved value of this collective promise as you can see from the above result.

💡 *Most of the static methods of the*Promise*class are generic as demonstrated in the below examples.*

There is one caveat with Promise.all. It implements the **fail-fast** mechanism which means if any of the input promises p1, p2, ... are rejected, pAll is rejected. If won’t wait for other pending promises to resolve.

## Promise.allSettled

The Promise.[allSettled](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled) static method is similar to Promise.all but unlike Promise.all, it waits until all the promises are settled (which means until they resolved or rejected). Hence the promise returned by Promise.allSettled will never reject (but we have added catch block in the below example anyway).

**Example**: promise-all-setlled.ts

*// A function that returns a delayed promise.*

const getPromise2 = (

*value*: number, *delay*: number, *fail*: boolean

): Promise<number> => {

    return new Promise<number>( (*resolve*, *reject*) => {

        setTimeout(

            () => fail ? reject(value) : resolve(value),

            delay

        );

    });

};

*/\**

*const allPromises2: Promise[*

*PromiseSettledResult<number>,*

*PromiseSettledResult<number>,*

*PromiseSettledResult<number>*

*]*

*\*/*

const allPromises2 = Promise.allSettled( [

getPromise2(0,0,false),         *// 0s.*

    getPromise2(1,2000,false),  *// 2s.*

    getPromise2(2,1000,true),   *// 1s.*

]);

*// Listen to allPromises2 resolution.*

console.time('settled in...');

allPromises2.then( (*value*) => {

*// (param) value: number[].*

    console.log('Resolved.', value);

}).catch((*error*) => {

*// (param) value: error.*

    console.log('Rejected.', error);

}).finally( () => {

    console.timeEnd('settled in...');

});

Working with promise.allSettled can be a little overwhelming as you can from the above program and its result. First of all, allSettle method returns a promise that resolves with an array of PromiseSettledResult<type> values when all promises are settled. The type is derived from the type of the input promise. The PromiseSettledResult type looks like below.

interface **PromiseFulfilledResult**<T> {  
 **status**: "fulfilled";  
 **value**: T; *// promise* ***resolved*** *value*  
}interface **PromiseRejectedResult** {  
 **status**: "rejected";  
 **reason**: any; *// promise* ***rejected*** *value*  
}type **PromiseSettledResult**<T> = **PromiseFulfilledResult**<T> | **PromiseRejectedResult**;

These types are provided by TypeScript’s standard library. So when a promise resolves, allSettled method converts its value into PromiseFulfilledResult shape and when it fails, it converts it to PromiseRejectedResult shape. That’s why when allSettled is resolved, it’s an array of objects in which each object has a shape of PromiseFulfilledResult or PromiseRejectedResult interface.

Since PromiseFulfilledResult is a union of PromiseFulfilledResult and PromiseRejectedResult that has common status property of literal data type, we can use it as a **discriminant** in the switch/case guard.

💡 *We have talked about*switch/case*type guard and discriminating unions in the*[***Type System***](https://medium.com/jspoint/typescript-type-system-81fdb84bba75)*lesson.*

**Example**: promise-all-setlled.ts

*// A function that returns a delayed promise.*

const getPromise3 = (

*value*: number, *delay*: number, *fail*: boolean

): Promise<number> => {

    return new Promise<number>((*resolve*, *reject*) => {

        setTimeout(

            () => fail ? reject(value) : resolve(value),

            delay

        );

    });

};

*/\**

*const allPromises2: Promise[*

*PromiseSettledResult<number>,*

*PromiseSettledResult<number>,*

*PromiseSettledResult<number>*

*]*

*\*/*

const allPromises3 = Promise.allSettled([

    getPromise3(0, 0, false),       *// 0s.*

    getPromise3(1, 2000, false),    *// 2s.*

    getPromise3(2, 1000, true),     *// 1s. (rejects).*

]);

*// Listen to allPromises3 resolution.*

console.time('settled in...');

allPromises3.then((*value*) => {

*// (param) value: number[].*

    console.log('Resolved.', value);

    value.forEach( *result* => {

        switch(result.status) {

            case 'fulfilled': {

*// result: PromiseFulfilledResult<number>.*

                console.log('success =>', result.value);

                break;

            }

            case 'rejected': {

*// result: PRomiseRejectedResult.*

                console.log('error =>', result.reason);

                break;

            }

        }

    });

}).catch((*error*) => {

*// (param) value: error.*

    console.log('Rejected.', error);

}).finally(() => {

    console.timeEnd('settled in...');

});

## Promise.race

The Promise.[race](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race) takes an array (iterable precisely) of promises and returns a new promise that resolves or rejects as soon as one of the input promises resolves or rejects. In other words, the promise returned by Promise.race is **settled** with the result of one of the input promises which **settles** quickly.

**Example**: promise-race.ts

*// A function that returns a delayed promise.*

const getPromise3 = (

*value*: number, *delay*: number, *fail*: boolean

): Promise<number> => {

    return new Promise<number>((*resolve*, *reject*) => {

        setTimeout(

            () => fail ? reject(value) : resolve(value),

            delay

        );

    });

};

*// const fastestPromise: Promise<number>*

const fastestPromise = Promise.race<number>([

    getPromise3(0, 500, false),     *// 0.5s.*

    getPromise3(1, 2000, false),    *// 2s.*

    getPromise3(2, 1000, true),     *// 1s. (rejects).*

]);

*// Listen to allPromises3 resolution.*

console.time('settled in...');

fastestPromise.then((*value*) => {

*// (param) value: number.*

    console.log('Resolved.', value);

}).catch((*error*) => {

*// (param) error: any.*

    console.log('Rejected.', error);

}).finally(() => {

    console.timeEnd('settled in...');

});

Unlike Promise.all or Promise.allSettled, this method only returns a single value of the first settled promise, hence the type of the returned promise is Promise<number> in the above case. Since the **1st** promise settled first among others, the then callback of the fastestPromise gets called after **500ms** with the value of the resolved promise.

💡 *The new*[*Promise.any*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/any)*() method has reached****Stage 4****of the ECMAScript proposal track. Promise.any is much like Promise.race but it waits until the first promise is****successful resolves****. This method throw an AggregateError exception if all the promises are rejected.*

## Async/Await keywords

Writing promises the normal way seems a bit difficult to manage. There is a lot of **unnecessary boilerplate** code involved. Normally, you have a function that processes some arguments and returns a promise that resolves or rejects with some meaningful data. Adding a promise creation mechanism using Promise constructor seems just **icky** at times.

JavaScript provide async keyword to turn a normal function into a function that returns a promise **implicitly**. Any value returned by this **async** function will be converted to an implicit promise that **resolves** with this value. If an error is thrown in this function, the returned promise is **rejected** with the error message.

The await keyword is used inside an async function to wait on a promise. If a promise has the await keyword before it, the function execution won’t proceed further until that promise is resolved.

**Example**: async-await.ts

*// Resolve with a random int b/w 0 and 9.*

const getRandomIntAsync: () => Promise<number> = *async* ()  => {

    return parseInt((Math.random() \* 10).toFixed(0));

};

*// Returns a promise of type "Promise<boolean>".*

*// const isEven: (answer: boolean) => Promise<boolean>.*

const isEven = *async* (*answer*: boolean) => {

*// const value: number.*

    const value = await getRandomIntAsync();

*// const isEven: boolean.*

    const isEven = value % 2 === 0;

*// return valis is "boolean".*

    return isEven === answer;

};

*// Listen to promise resolution.*

isEven(true).then((*value*) => {

*// (param) value: boolean.*

    console.log(value === true ? "Lucky :)" : "Unlucky :(", value);

}).catch( (*error*) => {

*// (parqm) error: any.*

    console.log('Rejected:', error);

}).finally( () => {

    console.log('Completed.');

});

In the above example, the getRandomInt is an async function since it has async keyword before the function expression. Since this function returns a promise which resolves with a number value, we have provided the correct type for the getRandomInt value (*shown below*).

const getRandomInt: **() => Promise<number>**

However, this is not necessary. TypeScript understands the async keyword, as well as looks at the return value type of the function to provide an implicit type for the function. Hence, the isEven constant has the below type.

const isEven: **(answer: boolean) => Promise<boolean>**

We have used await keyword inside the isEven function expression to wait for the resolution of the promise returned by the getRandomInt function. This means isEven function execution is basically going to halt until this promise is resolved, including the value assignment expression.

Since TypeScript knows about the type of the promise returned by the getRandomInt function, the value constant has the number type. The isEven async function returns a promise that resolved with a boolean value.

You might wonder, how the promise returned by a async function is going to reject? Well, I have explained this in my [**other article**](https://medium.com/jspoint/javascript-promises-and-async-await-as-fast-as-possible-d7c8c8ff0abc) on Promises but in a nutshell, the promise returned by an async function rejects if an error is thrown inside the async function.

**Example**: async-await-reject.ts

*// Resolve with a random int b/w 0 and 9.*

const getRandomIntAsync2: () => Promise<number> = *async* ()  => {

    return parseInt((Math.random() \* 10).toFixed(0));

};

*// Returns a promise of type "Promise<boolean>". Resolve with a boolean value.*

*// const isEven2: (answer: boolean) => Promise<boolean>.*

const isEven2 = *async* (*answer*: boolean) => {

*// const value: number.*

    const value = await getRandomIntAsync2();

*// Reject if the value is 0.*

    if(value === 0) {

        throw new Error('Can\'t work with 0.');

    }

*// const isEven: boolean.*

    const isEven = value % 2 === 0;

*// return valis is "boolean".*

    return isEven === answer;

};

*// Listen to promise resolution.*

isEven2(true).then((*value*) => {

*// (param) value: boolean.*

    console.log(value === true ? "Lucky :)" : "Unlucky :(", value);

}).catch( (*error*) => {

*// (parqm) error: any.*

    console.log('Rejected:', error);

}).finally( () => {

    console.log('Completed.');

});

In the above example, we have done some modifications to the previous example. We have added a check inside isEven function that checks if the value is 0 then throws a generic JavaScript error.

In this scenario, there is a chance that the promise returned by the isEven function may get rejected. This will be caught by the catch method and it will contain the error that was thrown in the async function.

We have provided the Error type for the error argument of the callback function of the catch method to assert the default any type to Error type since we know the rejection value type beforehand.

Since an async function uses throw keyword to throw an error in order to reject the promise, it raises another question. What will happen if another async function is awaiting on the promise if that promise gets rejected?

The short answer is, errors are bubbled up in the async functions. This means if the promise we are awaiting (using await keyword) rejects, JavaScript will throw an error in the current async function and nothing below that await keyword will execute. This goes on if the current async function is also being used inside another async function.

To check if the awaited promise was rejected, we use try/catch. If we await a promise inside a try block, we will know if the promise was rejected in the catch block. The argument to catch block is the rejection value.

💡 *FYI, you can also await a promise returned by the*Promise*constructor. There won’t be any difference whatsoever. If it rejects, it will result in an error.*

**Example**: async-await-try-catch.ts

*// Resolve with a random int b/w 0 and 9.*

*// const getRandomIntAsync3: () => Promise<number>.*

const getRandomIntAsync3: () => Promise<number> = *async* () => {

    const value = parseInt((Math.random() \* 10).toFixed(0));

*// Reject if value is 0.*

    if (value === 0) {

        throw new Error('Can\'t work with 0.');

    }

    return value;

};

*// Returns a promise of type "Promise<boolean>". Resolve with a boolean value.*

*// const isEven2: (answer: boolean) => Promise<boolean>.*

const isEven3 = *async* (*answer*: boolean) => {

    try {

*// const value: number.*

        const value = await getRandomIntAsync3();

*// const isEven: boolean.*

        const isEven = value % 2 === 0;

*// return valis is "boolean".*

        return isEven === answer;

    } catch( e) {

        console.log('getRandomIntAsync3 rejection:', e.message);

        return false;

    }

};

*// Listen to promise resolution.*

isEven3(true).then((*value*) => {

*// (param) value: boolean.*

    console.log(value === true ? "Lucky :)" : "Unlucky :(", value);

}).catch((*error*: Error) => {

*// (parqm) error: Error.*

    console.log('Rejected:', error);

}).finally(() => {

    console.log('Completed.');

});

In the above example, the promise returned by the getRandomInt can get rejected. Hence inside the isEven async function, we have wrapped the awaiting logic inside the try block. If the code inside try block throws an error, the catch block will be executed with that error.

In this example, the isEven async function always returns a promise that never rejects. If you want to reject this promise, you need to throw an error manually using throw keyword within the catch block

# Angular – HttpClient

<https://www.telerik.com/blogs/angular-basics-how-to-use-httpclient>

<https://www.techiediaries.com/angular-by-example-httpclient-get/>

## Why Do We Need HttpClient?

The front-end of applications communicate with back-end services to get or send the data over HTTP protocol using either XMLHttpRequest interface or fetch API. This communication is done in Angular with the help of HttpClient.

## What Is HttpClient?

HttpClient is a built-in service class available in the @angular/common/http package. It has multiple signature and return types for each request. It uses the RxJS observable-based APIs, which means it returns the observable and what we need to subscribe it. This API was developed based on XMLHttpRequest interface exposed by browsers.

## Features of HttpClient

* Provides typed request and response objects
* Contains testability features
* Intercepts request and response
* Supports RxJS observable-based APIs
* Supports streamlined error handling
* Performs the GET, POST, PUT, DELETE operations

## What Is an RxJS Observable?

An observable is a unique object similar to Promise and it helps to manage async code. It’s not from the JavaScript language, so to use it we need the most popular observable library, called RxJS (Reactive Extension for JavaScript). RxJS is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code. Angular uses observables as an interface to handle the common asynchronous operations.

Following are the points to consider when we use HttpClient and it returns the observables :

* When we subscribe, it will initiate the request, otherwise nothing happens.
* When the get() request returns successful, the observable emits the result and is then complete.
* When the get() request fails, the observable emits the error.

## How To Use HttpClient in Angular?

Let’s see how to use this HttpClient module in an Angular application. This module is already included in the application when we create the application in Angular. Follow the steps below to use it:

**Source**: /Angular Demos/httpclient-app

**Step 1:** I have created the application with the help of angular-cli command ng new httpclient-app.

**Step 2:** Import or configure the HttpClientModule into the app.module.ts file as shown below:

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { HttpClientModule } from '@angular/common/http';

@NgModule({

declarations: [AppComponent],

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule //imported the module

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Step 3:** You can directly use the HttpClient in your component, but its best to access it via the service. We are creating a new service with the help of angular-cli command ng generate service service-name. You will see code in that service as below:

ng g service services/HttpService

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class HttpService {

constructor() { }

}

**Step 4:** Inject the HttpClient in the service created in the previous step. Then you can see code as below:

import { HttpClient } from '@angular/common/http';

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class HttpService {

constructor(private http: HttpClient) { }

}

**Step 5:** In this step we are going to fetch the data from the server with the help of HTTP GET request. For that, we are adding one method in the service name as getPosts—that method we are calling in the component.

import { HttpClient } from '@angular/common/http';

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class HttpService {

private url = '<https://my-json-server.typicode.com/JSGund/XHR-Fetch-Request-JavaScript/posts>';

constructor(private http: HttpClient) { }

getPosts() {

return this.http.get(this.url);

}

}

In the above code we have added the method getPosts and placed HTTP GET request, and passed the one parameter with the request that is nothing but the End-point-url. It’s always best practice to keep the constants in a separate file and access them from there—it’s easy to share them and reference them for modification when there is a single place to reflect where they are used.

**Step 6:** Let’s understand the **HTTP GET** request and its request and response objects. The HTTP GET request has around 15 different types of methods to use.

get<T>(url: string, options?: { headers?: [HttpHeaders];

context?: [HttpContext];

observe?: "body";

params?: [HttpParams];

reportProgress?: boolean;

responseType?: "json";

withCredentials?: boolean;

}): Observable<T>

***Parameters***

* url – It is the service/API endpoint URL of type string.
* options – It is used to configure the HTTP request. It is optional and of type object, and its default value is undefined.

options:

{

headers?: [HttpHeaders],

observe?: 'body' | 'events' | 'response',

params?: [HttpParams],

reportProgress?: boolean,

responseType?: 'arraybuffer'|'blob'|'json'|'text',

withCredentials?: boolean,

}

Below two are important options properties:

* observe: How much of the response to return.
* responseType: The return data format.

***Returns***HTTP GET returns an observable of the HttpResponse.

**Step 7:** In this step we are going to use the getPosts method in the component. For that, first we need to inject the created service into our component and access the method as shown below:

import { Component } from '@angular/core';

import { HttpService } from './http.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Article by Jeetendra';

posts : any;

constructor(private httpService: HttpService) { }

ngOnInit() {

this.httpService.getPosts().subscribe(

(response) => { this.posts = response; },

(error) => { console.log(error); });

}

}

In the above code we have injected the service first in constructor then the important thing is we call the getPosts method and subscribe to it. Whenever we get the response from this subscribe method, it will be a list of object containing id, title, path, as shown below:

***Response***

[

{id: 1, title: "Angular Localization Using ngx-translate", path: "<https://www.telerik.com/blogs/angular-localization-using-ngx-translate>"},

{id: 2, title: "How to Use the Navigation or Shadow Property in Entity Framework Core", path: "<https://www.telerik.com/blogs/how-to-use-the-navigation-or-shadow-property-in-entity-framework-core>"},

{id: 3, title: "Getting Value from appsettings.json in .NET Core", path: "<https://www.telerik.com/blogs/how-to-get-values-from-appsettings-json-in-net-core>"},

{id: 4, title: "Embedding Beautiful Reporting into Your ASP.NET MVC Applications", path: "<https://www.telerik.com/blogs/embedding-beautiful-reporting-asp-net-mvc-applications>"},

{id: 5, title: "Select Tag Helper in ASP.NET Core MVC", path: "<https://www.telerik.com/blogs/select-tag-helper-asp-net-core-mvc>"}

]

We have declared the property as posts and assigned the response we get in the subscribe, then iterated that in HTML with the help of the \*ngFor directive as the code below shows:

<div>

<li \*ngFor="let post of posts">

<a href="{{post.path}}">

<span style="font-size: 20px;text-align: center;">{{post.title}}

</span>

</a>

</li>

</div>

**Step 8:** Finally, we have implemented our first HTTP request, that GET. Run the Angular application with help of angular-cli command ng serve, and you will get a message like, “Angular Live Development Server is listening on localhost:4200. Open your browser on [http://localhost:4200/.”](http://localhost:4200/.%E2%80%9D) Once you open the URL in your browser, you will get an output like the image below.

***Output***

Graphical user interface, text, application, chat or text message

Description automatically generated

In the above output you will see the list of posts written by me.

## Another Example

<https://medium.com/egen/using-angular-httpclient-the-right-way-60c65146e5d9>

**Source**: /Angular Demos/httpclient-app

### Service Setup

Now we can set up the **HttpClient** to make the API calls, we will be retrieving the users from the endpoint <https://jsonplaceholder.typicode.com/users>. It is pretty straight forward to make the API call, so let us set up the appropriate service to do so.

Creating services with the help of angular cli is one of the best ways to do so. You can create one manually but it’s always easier to generate it using the CLI as it uses the latest syntax and creates the service, its spec and adds it to the nearest module as a provider.

ng generate service services/user

Once the service is set up, we just need to inject the **HttpClient** and we are ready to make the API call.

### Making the API call

To make the API call, we can simply inject the **HttpClientModule** module in the main application module as follows:

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Next we add the method necessary to make the API call for getting the users from the URL <https://jsonplaceholder.typicode.com/users> as shown below:

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

@Injectable({

providedIn: 'root'

})

export class UserService {

constructor(private http: HttpClient) {}

getUsers() {

return this.http.get('https://jsonplaceholder.typicode.com/users');

}

}

Last and final piece of code necessary to get the users list is to make the call to service method. You can add this logic to either the constructor or the OnInit method based on your preference. We will go with the constructor for now to keep the example simple.

import { Component } from '@angular/core';

import { UserService } from './services/user.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'app';

constructor(private userService: UserService) {

this.userService.getUsers()

.subscribe((resp) => {

console.log(resp);

});

}

}

ng serve

Once the application is open on the browser, open the console and you can see the list fo users logged as below:

Text

Description automatically generated

As expected, this works and logs the list of users that were retrieved, but there is no type safety being enforced. One of the great advantages with typescript projects is that we can leverage interface.

### User Interface

Now that we have the API call working as expected, let us create the user interface which we will use in the service later on. We will once again be using the angular cli to create the interface:

ng generate interface interfaces/user

We can now add the fields necessary to the interface that we just created.

export interface User {

id: string;

name: string;

username: string;

email: string;

address: {

street: string;

suite: string;

city: string;

zipcode: number;

geo: {

lat: number;

lng: number;

};

};

phone: string;

website: string;

company: {

name: string;

catchPhrase: string;

bs: string;

};

age: number;

}

### Updated Component

Based on the API response we were expecting, we have created the interface for the User object. We will now be embedding the User interface in all our service calls which is an easy change.

In the service:

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { User } from '../interfaces/user';

import { Observable } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class UserService {

constructor(private http: HttpClient) {}

getUsers(): Observable<User[]> {

return this.http.get<User[]>('https://jsonplaceholder.typicode.com/users');

}

}

and in the component:

import { Component } from '@angular/core';

import { UserService } from './services/user.service';

import { User } from './interfaces/user';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'app';

constructor(private userService: UserService) {

this.userService.getUsers()

.subscribe((resp: User[]) => {

console.log(resp);

});

}

}

It still works as expected:

Text

Description automatically generated

Now one might wonder, what is the big deal with the **interface**, its not adding anything extra, not enforcing any rules, not adding default or new values etc. In short, Interface does not do much except enforce the structure of the User that we are are consuming in the app component.

Now consider the case where we want to filter out some fields from the response of the API or if we want to add some custom validations or modifications to the User. This is where we upgrade our User interface to a class and add all the properties or customizations that we need.

First, let us rename the User interface to BaseUser. We do this so that our new User class inherits from the BaseUser thus keeping the type check in place.

export interface BaseUser {

id: string;

name: string;

username: string;

email: string;

address: {

street: string;

suite: string;

city: string;

zipcode: number;

geo: {

lat: number;

lng: number;

};

};

phone: string;

website: string;

company: {

name: string;

catchPhrase: string;

bs: string;

};

}

Create a class named User with

ng g class classes/User

Update User class which implements the BaseUser

import { BaseUser } from "../interfaces/baseuser";

export class User implements BaseUser {

    id: string;

    name: string;

    username!: string;

    email!: string;

    address!: {

      street: string;

      suite: string;

      city: string;

      zipcode: number;

      geo: {

        lat: number;

        lng: number;

      };

    };

    phone!: string;

    website!: string;

    company!: {

      name: string;

      catchPhrase: string;

      bs: string;

    };

    age!: number;

    constructor(user: BaseUser) {

      this.id = user.id;

      this.name = user.name.toUpperCase();

    }

  }

For now, we are returning only the id and name so that we can test the changes and get an idea of how it works. Next, we need a way to invoke the this new class that we have created. This can be done in two ways:

1. Add an Http Interceptor to change the user inflight, but might apply your change to all the calls.
2. Invoke the User class on each of the record and update the structure of the object.

In this example, we want this change to apply only on our app component so we will go with approach #2.

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

import { User } from '../interfaces/user';

import { map } from 'rxjs/operators';

@Injectable({

providedIn: 'root'

})

export class UserService {

constructor(private http: HttpClient) {}

getUsers(): Observable<User[]> {

return this.http

.get<User[]>('https://jsonplaceholder.typicode.com/users')

.pipe(map((users: User[]) => users.map(user => new User(user))));

}

}

As you can see, we chained a simple pipe to our get call to capture the Observable and use the map operator from rxjs to iterate over the response and modify each record as new User class instance.

### Result

And that is it. With these changes in, we can now see that the response that we get in our app.componentis of the type User. Now you can add all sorts of additional validations and modifications to the User.

Graphical user interface, text, application, email

Description automatically generated

# Angular – HttpClient – CRUD

<https://www.pluralsight.com/guides/posting-deleting-putting-data-angular>

**Source**: Angular Demos/httpclient-app

ng g service services/HttpCrudDemo –skip-tests=true

import { HttpClient } from '@angular/common/http';

import { Injectable } from '@angular/core';

@Injectable({

  providedIn: 'root'

})

export class HttpCrudDemoService {

  url: String = "";

  constructor(private http: HttpClient) {

    this.url = "https://jsonplaceholder.typicode.com";

  }

  public getAllPosts()  {

    console.log("HTTP CRUD DEMO...");

    let endPoints = "/posts";

    return this.http.get(this.url + endPoints);

  }

  public getPostById(postId: number) {

    //let id: number = postId;

    let id: number = 1;

    console.log("Got id: " +  id);

    let endpoints = "/posts/" + id;

    return this.http.get(this.url + endpoints);

  }

  public addPost(postData: Object) {

    let endpoints = "/posts";

    return this.http.post(this.url + endpoints, postData);

  }

  public updatePost(postData: Object) {

    let endPoints = "/posts/1"

    return this.http.put(this.url + endPoints, postData);

  }

  public deletePost() {

    let endPoints = "/posts/1"

    return this.http.delete(this.url + endPoints);

  }

}

The app.component.ts:

import { Component } from '@angular/core';

import { HttpCrudDemoService } from './services/http-crud-demo.service';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'httpclient-app';

  posts: any;

  postId: number = 0;

  constructor(private crudService: HttpCrudDemoService) {

  }

  ngOnInit() {

  }

  getAllPosts() {

    this.crudService.getAllPosts().subscribe(

      (response) => { this.posts = response; },

      (error) => { console.log(error); }

    )

  }

  getPostById() {

    this.crudService.getPostById(this.postId).subscribe(

      (response) => { console.log(response); },

      (error) => { console.log(error); }

    )

  }

  addPost() {

    this.crudService.addPost({"userID": 1, "id": 999, "title": "Ajay", "body": "test by ajay"}).subscribe(

      (response) => { console.log(response); },

      (error) => { console.log(error); }

    )

  }

  updatePost() {

    this.crudService.updatePost({"userID": 1, "id": 101, "title": "Ajay", "body": "test by ajay singala"}).subscribe(

      (response) => { console.log(response); },

      (error) => { console.log(error); }

    )

  }

  deletePost() {

    this.crudService.deletePost().subscribe(

      (response) => { console.log(response); },

      (error) => { console.log(error); }

    )

  }

}

Ad the following in app.module.ts:

:

import { FormsModule } from '@angular/forms';

:

  imports: [

:

    FormsModule

  ],

:

The app.component.html:

<div>

  <li \*ngFor="let post of posts">

       <a  href="{{post.path}}">

           <span  style="font-size: 20px;text-align: center;">{{post.title}}

            </span>

      </a>

  </li>

</div>

<div>

  <button (click)="getAllPosts();">Get All Posts</button>

  <br>

  <input type="text" [(ngModel)]="postId">

  <button (click)="getPostById();">Get Post By Id</button>  {{ postId }}

  <br>

  <button (click)="addPost();">Add Post</button>

  <br>

  <button (click)="updatePost();">Update Post</button>

  <br>

  <button (click)="deletePost();">Delete Post</button>

</div>

<router-outlet></router-outlet>

Check the developer console for results.

# Angular – HttpClient – Calling .NET Core API (with CORS)

**Source**: Angular Demos\todo-app

**Repo**: <https://github.com/AjaySingala/ng-ts-node/tree/main/Angular%20Demos/todo-app>

# Angular – Pipes

<https://www.tutorialspoint.com/angular8/angular8_pipes.htm>

<https://www.simplilearn.com/tutorials/angular-tutorial/angular-pipes>

Pipes are referred as filters. It helps to transform data and manage data within interpolation, denoted by {{ | }}. It accepts data, arrays, integers and strings as inputs which are separated by ‘|’ symbol. This chapter explains about pipes in detail.

**Source: /Andgular Demos/pipes-app**

## Create the app and component

ng new pipes-app

ng g c components/pipes

## Adding Parameters

Create a date method in your **pipes.component.ts**file.

export class PipesComponent {

presentDate = new Date();

}

Now, add the below code in your pipes.component.html file.

<div>

Today's date :- {{presentDate}}

</div>

Now, run the application, it will show the following output −

Today's date :- Mon Jun 15 2020 10:25:05 GMT+0530 (IST)

Here, Date object is converted into easily readable format.

### Add Date pipe

Let’s add date pipe in the above html file.

<div>

Today's date :- {{presentDate | date }}

</div>

You could see the below output −

Today's date :- Jun 15, 2020

### Parameters in Date

We can add parameter in pipe using : character. We can show short, full or formatted dates using this parameter. Add the below code in **pipes.component.html** file.

<div>

short date :- {{presentDate | date:'shortDate' }} <br/>

Full date :- {{presentDate | date:'fullDate' }} <br/>

Formatted date:- {{presentDate | date:'M/dd/yyyy'}} <br/>

Hours and minutes:- {{presentDate | date:'h:mm'}}

</div>

You could see the below response on your screen −

short date :- 6/15/20

Full date :- Monday, June 15, 2020

Formatted date:- 6/15/2020

Hours and minutes:- 12:00

## Chained pipes

We can combine multiple pipes together. This will be useful when a scenario associates with more than one pipe that has to be applied for data transformation.

In the above example, if you want to show the date with uppercase letters, then we can apply both **Date** and **Uppercase** pipes together.

<div>

Date with uppercase :- {{presentDate | date:'fullDate' | uppercase}} <br/>

Date with lowercase :- {{presentDate | date:'medium' | lowercase}} <br/>

</div>

You could see the below response on your screen −

Date with uppercase :- MONDAY, JUNE 15, 2020 Date with lowercase :- jun 15, 2020, 12:00:00 am

Here, Date, Uppercase and Lowercase are pre-defined pipes. Let’s understand other types of built-in pipes in next section.

## Built-in Pipes

Angular supports the following built-in pipes. We will discuss one by one in brief.

### AsyncPipe

If data comes in the form of observables, then **Async pipe** subscribes to an observable and returns the transmitted values.

import { Observable, Observer } from 'rxjs';

export class PipesComponent implements OnInit {

timeChange = new Observable<string>((observer: Observer<string>) => {

setInterval(() => observer.next(new

Date().toString()), 1000);

});

}

Here, the **Async** pipe performs subscription for time changing in every one seconds and returns the result whenever gets passed to it. Main advantage is that, we don’t need to call subscribe on our timeChange and don’t worry about unsubscribe, if the component is removed.

Add the below code inside your **pipes.component.html**.

<div>

Seconds changing in Time: {{ timeChange | async }}

</div>

Now, run the application, you could see the seconds changing on your screen.

### CurrencyPipe

It is used to convert the given number into various countries currency format. Consider the below code in **pipes.component.ts** file.

import { Component, OnInit } from '@angular/core'; @Component({

selector: 'app-pipes',

template: `

<div style="text-align:center">

<h3> Currency Pipe</h3>

<p>{{ price | currency:'EUR':true}}</p>

<p>{{ price | currency:'INR' }}</p>

</div>

`,

styleUrls: ['./pipes.component.scss']

})

export class PipesComponent implements OnInit {

price : number = 20000; ngOnInit() {

}

}

You could see the following output on your screen −

Currency Pipe

€20,000.00

₹20,000.00

### SlicePipe

Slice pipe is used to return a slice of an array. It takes index as an argument. If you assign only start index, means it will print till the end of values. If you want to print specific range of values, then we can assign start and end index.

We can also use negative index to access elements. Simple example is shown below −

**pipes.component.ts**

import { Component, OnInit } from '@angular/core'; @Component({

selector: 'app-pipes,

template: `

<div>

<h3>Start index:- {{Fruits | slice:2}}</h3>

<h4>Start and end index:- {{Fruits | slice:1:4}}</h4>

<h5>Negative index:- {{Fruits | slice:-2}}</h5>

<h6>Negative start and end index:- {{Fruits | slice:-4:-2}}</h6>

</div>

`,

styleUrls: ['./pipes.component.scss']

})

export class PipesComponent implements OnInit {

Fruits = ["Apple","Orange","Grapes","Mango","Kiwi","Pomegranate"];

ngOnInit() {

}

}

Now run your application and you could see the below output on your screen −

Start index:- Grapes,Mango,Kiwi,Pomegranate

Start and end index:- Orange,Grapes,Mango

Negative index:- Kiwi,Pomegranate

Negative start and end index:- Grapes,Mango

Here,

* **{{Fruits | slice:2}}** means it starts from second index value Grapes to till the end of value.
* **{{Fruits | slice:1:4}}** means starts from 1 to end-1 so the result is one to third index values.
* **{{Fruits | slice:-2}}** means starts from -2 to till end because no end value is specified. Hence the result is Kiwi, Pomegranate.
* **{{Fruits | slice:-4:-2}}** means starts from negative index -4 is Grapes to end-1 which is -3 so the result of index[-4,-3] is Grapes, Mango.

### DecimalPipe

It is used to format decimal values. It is also considered as CommonModule. Let’s understand a simple code in **pipes.component.ts** file,

import { Component, OnInit } from '@angular/core'; @Component({

selector: 'app-pipes',

template: `

<div style="text-align:center">

<h3>Decimal Pipe</h3>

<p> {{decimalNum1 | number}} </p>

<p> {{decimalNum2 | number}} </p>

</div>

`,

styleUrls: ['./pipes.component.scss']

})

export class PipesComponent implements OnInit {

decimalNum1: number = 8.7589623;

decimalNum2: number = 5.43;

ngOnInit() {

}

}

You could see the below output on your screen −

Decimal Pipe

8.759

5.43

### Formatting values

<https://www.angularjswiki.com/angular/angular-decimal-pipe/>

We can apply string format inside number pattern. It is based on the below format −

number:"{minimumIntegerDigits}.{minimumFractionDigits} - {maximumFractionDigits}"

Angular decimal pipe acceps two parameters

1. decimal digit info
2. locale

{{ **numeric\_value** | **number** [ : digitsInfo [ : locale ] ] }}

| **Parameter** | **Description** |
| --- | --- |
| digitsInfo | Decimal representation string in the following format: {minimumIntegerDigits}.{minimumFractionDigits}-{maximumFractionDigits} |
| minimumIntegerDigits | Represents minimum number of integer digits before the decimal point. Default value is 1. |
| minimumFractionDigits | Represents minimum number of digits after the decimal point. Default value is 0. |
| maximumFractionDigits | Represents maximum number of digits after the decimal point. Default value is 3. |
| locale | represents locale format rules to use. Default value is our project locale (en\_US) if set or undefined.Optional parameter |

decimal\_value: number = 5.123456789;

**No Parameters:**

{{ decimal\_value | number }}

// No Parameters: 5.123

So, the above code is same as:

{{ decimal\_value | number:'1.0-3' }}

Now we will pass digit information parameter to decimal pipe and see how it works

Digit Info Parameter (3.1-5):

{{ decimal\_value | number:'3.1-5' }}

In the above code we are instructing decimal pipe to show at least 3 integer values before decimal points and minimum 1 fraction digit, maximum 5 fraction digits.

Digit Info Parameter (3.1-5): 005.12346

As our decimal value contains only single digit before decimal point two extra zeroes are padded two satisfy given digit info.

Let’s apply the above format in our code,

@Component({

template: `

<div style="text-align:center">

<p> Apply formatting:- {{decimalNum1 | number:'3.1'}} </p>

<p> Apply formatting:- {{decimalNum1 | number:'2.1-4'}} </p>

</div>

`,

})

Here,

**{{decimalNum1 | number:’3.1’}}** means three decimal place and minimum of one fraction but no constraint about maximum fraction limit. It returns the following output −

Apply formatting:- 008.759

**{{decimalNum1 | number:’2.1-4’}}** means two decimal places and minimum one and maximum of four fractions allowed so it returns the below output −

Apply formatting:- 08.759

#### Angular decimal pipe example with country locale

To display numbers according to country locale format rules, We have to pass country locale code as a second parameter to angular decimal pipe.

To display number in french locale in Angular use locale code ‘fr’ as parameter as shown below.

<p>

    Decimal pipe with locale france:

    {{decimal\_value | number:'4.5-5':'fr'}}

    <br>

    {{decimal\_value | currency: 'CAD': 'symbol': '4.5-5':'fr'}}

</p>

If you execute the above code, you will get following error in console:

InvalidPipeArgument: 'Missing locale data for the locale "fr".'

for pipe 'DecimalPipe'

Because in our application we don’t have locale information for ‘fr’.

To fix this, we need to register the locale information. Follow the below steps to use Angular Currency Pipe with locale.

1. Import the registerLocaleData from @angular/common
2. Import locale Information from @angular/common/locales/fr.
3. And Finally register the information using registerLocaleData method.

Add the following code to **pipes.component.ts** and then take a look at the result:

import { registerLocaleData } from '@angular/common';

import localeFr from '@angular/common/locales/fr';

registerLocaleData(localeFr, 'fr');

### PercentPipe

It is used to format number as percent. Formatting strings are same as DecimalPipe concept. Simple example is shown below −

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-pipes,

template: `

<div style="text-align:center">

<h3>Decimal Pipe</h3>

<p> {{percentNum | percent:'2.2'}} </p>

</div>

`,

styleUrls: ['./pipes.component.scss']

})

export class PipesComponent {

percentNum: number = 0.8178;

}

You could see the below output on your screen −

Decimal Pipe

81.78%

### JsonPipe

It is used to transform a JavaScript object into a JSON string. Add the below code in **pipes.component.ts** file as follows −

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-pipes',

template: `

<div style="text-align:center">

<p ngNonBindable>{{ jsonData }}</p> (1)

<p>{{ jsonData }}</p>

<p ngNonBindable>{{ jsonData | json }}</p>

<p>{{ jsonData | json }}</p>

</div>

`,

styleUrls: ['./pipes.component.scss']

})

export class PipesComponent {

jsonData = { id: 'one', name: { username: 'user1' }}

}

Now, run the application, you could see the below output on your screen −

{{ jsonData }}

(1)

[object Object]

{{ jsonData | json }}

{ "id": "one", "name": { "username": "user1" } }

## Creating custom pipe

As we have seen already, there is a number of pre-defined Pipes available in Angular 8 but sometimes, we may want to transform values in custom formats. This section explains about creating custom Pipes.

Create a custom Pipe using the below command −

ng g pipe pipes/DigitCount

Let’s create a logic for counting digits in a number using Pipe. Open **digitcount.pipe.ts** file and add the below code −

import { Pipe, PipeTransform } from '@angular/core'; @Pipe({

name: 'digitCount'

})

export class DigitCountPipe implements PipeTransform {

transform(num : number) : number {

return num.toString().length;

}

}

Now, we have added logic for count number of digits in a number. Let’s add the final code in **pipes.component.ts** file as follows −

import { Component, OnInit } from '@angular/core'; @Component({

selector: 'app-pipes',

template: `

<div>

<p> DigitCount Pipe </p>

<h1>{{ digits | digitCount }}</h1>

</div>

`,

styleUrls: ['./pipes.component.scss']

})

export class PipesComponent implements OnInit {

digits : number = 100;

ngOnInit() {

}

}

Now, run the application, you could see the below response −

DigitCount Pipe

3

# Angular – Routing

<https://www.smashingmagazine.com/2018/11/a-complete-guide-to-routing-in-angular/>

<https://www.geeksforgeeks.org/routing-in-angular-9-10/>

**Source Code: Angular Demos/router-app**

## Introducing Angular Router

[Angular Router](https://angular.io/guide/router) is a powerful JavaScript router built and maintained by the Angular core team that can be installed from the @angular/router package. It provides a complete routing library with the possibility to have multiple router outlets, different path matching strategies, easy access to route parameters and route guards to protect components from unauthorized access.

The Angular router is a core part of the Angular platform. It enables developers to build Single Page Applications with multiple views and allow navigation between these views.

Let’s now see the essential Router concepts in more details.

## THE ROUTER-OUTLET

The [Router-Outlet](https://angular.io/api/router/RouterOutlet) is a directive that’s available from the router library where the Router inserts the component that gets matched based on the current browser’s URL. You can add multiple outlets in your Angular application which enables you to implement advanced routing scenarios.

<router-outlet></router-outlet>

Any component that gets matched by the Router will render it as a sibling of the Router outlet.

## ROUTES AND PATHS

Routes are definitions (objects) comprised from at least a path and a component (or a [redirectTo](https://angular.io/api/router/Route#redirectTo) path) attributes. The path refers to the part of the URL that determines a unique view that should be displayed, and component refers to the Angular component that needs to be associated with a path. Based on a route definition that we provide (via a static RouterModule.forRoot(routes) method), the Router is able to navigate the user to a specific view.

Each [Route](https://angular.io/api/router/Route) maps a URL path to a component.

The path can be empty which denotes the default path of an application and it’s usually the start of the application.

The path can take a **wildcard** string (\*\*). The router will select this route if the requested URL doesn’t match any paths for the defined routes. This can be used for displaying a “Not Found” view or redirecting to a specific view if no match is found.

This is an example of a route:

{ path: 'contacts', component: ContactListComponent}

If this route definition is provided to the Router configuration, the router will render ContactListComponent when the browser URL for the web application becomes /contacts.

## ROUTE MATCHING STRATEGIES

The Angular Router provides different route matching strategies. The default strategy is simply checking if the current browser’s URL is prefixed with the path.

For example our previous route:

{ path: 'contacts', component: ContactListComponent}

Could be also written as:

{ path: 'contacts',pathMatch: 'prefix', component: ContactListComponent}

The patchMath attribute specifies the matching strategy. In this case, it’s **prefix** which is the default.

The second  matching strategy is **full**. When it’s specified for a route, the router will check if the the path is exactly **equal** to the path of the current browser’s URL:

{ path: 'contacts',pathMatch: 'full', component: ContactListComponent}

## ROUTE PARAMS

Creating routes with parameters is a common feature in web apps. Angular Router allows you to access parameters in different ways:

* Using the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) service,
* Using the [ParamMap](https://angular.io/api/router/ParamMap) observable available starting with v4.

You can create a route parameter using the **colon** syntax. This is an example route with an id parameter:

{ path: 'contacts/:id', component: ContactDetailComponent}

## ROUTE GUARDS

A route guard is a feature of the Angular Router that allows developers to run some logic when a route is requested, and based on that logic, it allows or denies the user access to the route. It’s commonly used to check if a user is logged in and has the authorization before he can access a page.

You can add a route guard by implementing the [CanActivate](https://angular.io/api/router/CanActivate) interface available from the @angular/router package and extends the canActivate() method which holds the logic to allow or deny access to the route. For example, the following guard will always allow access to a route:

class MyGuard implements CanActivate {

canActivate() {

return true;

}

}

You can then protect a route with the guard using the canActivate attribute:

{ path: 'contacts/:id, canActivate:[MyGuard], component: ContactDetailComponent}

## NAVIGATION DIRECTIVE

The Angular Router provides the routerLink directive to create navigation links. This directive takes the path associated with the component to navigate to. For example:

<a [routerLink]="'/contacts'">Contacts</a>

## MULTIPLE OUTLETS AND AUXILIARY ROUTES

Angular Router supports multiple outlets in the same application.

A component has one associated primary route and can have auxiliary routes. Auxiliary routes enable developers to navigate multiple routes at the same time.

To create an auxiliary route, you’ll need a named router outlet where the component associated with the auxiliary route will be displayed.

<router-outlet></router-outlet>

<router-outlet name="outlet1"></router-outlet>

* The outlet with no name is the primary outlet.
* All outlets should have a name except for the primary outlet.

You can then specify the outlet where you want to render your component using the outlet attribute:

{ path: "contacts", component: ContactListComponent, outlet: "outlet1" }

## Creating an Angular Project with Routing

ng new router-app

### Creating A Fake Back-End Service

Since we don’t have a real back-end to interact with, we’ll create a fake back-end using the [angular-in-memory-web-api](https://github.com/angular/in-memory-web-api) library which is an in-memory web API for Angular demos and tests that emulates CRUD operations over a REST API.

It works by intercepting the HttpClient requests sent to the remote server and redirects them to a local in-memory data store that we need to create.

To create a fake back-end, we need to follow the next steps:

1. First, we install the angular-in-memory-web-api module,
2. Next, we create a service which returns fake data,
3. Finally, configure the application to use the fake back-end.

In your terminal run the following command to install the angular-in-memory-web-api module from npm:

$ npm install --save angular-in-memory-web-api

Next, generate a back-end service using:

$ ng g s services/backend

Open the src/app/services/backend.service.ts file and import InMemoryDbService from the angular-in-memory-web-api module:

import {InMemoryDbService} from 'angular-in-memory-web-api'

The service class needs to implement InMemoryDbService and then override the createDb() method:

@Injectable({

providedIn: 'root'

})

export class BackendService implements InMemoryDbService{

constructor() { }

createDb(){

let contacts = [

{ id: 1, name: 'Contact 1', email: 'contact1@email.com' },

{ id: 2, name: 'Contact 2', email: 'contact2@email.com' },

{ id: 3, name: 'Contact 3', email: 'contact3@email.com' },

{ id: 4, name: 'Contact 4', email: 'contact4@email.com' }

];

return {contacts};

}

}

We simply create an array of contacts and return them. Each contact should have an id.

Finally, we simply need to import InMemoryWebApiModule into the app.module.ts file, and provide our fake back-end service.

import { InMemoryWebApiModule } from “angular-in-memory-web-api”;

import { BackendService } from “../app/services/backend.service”;

/\* ... \*/

@NgModule({

declarations: [

/\*...\*/

],

imports: [

/\*...\*/

InMemoryWebApiModule.forRoot(BackendService)

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Also import HttpClientModule:

:

import { HttpClientModule } from '@angular/common/http';

:

  imports: [

    BrowserModule,

    AppRoutingModule,

    HttpClientModule,

    InMemoryWebApiModule.forRoot(BackendService)

  ],

:

export class AppModule { }

Next create a ContactService which encapsulates the code for working with contacts:

$ ng g s services/contact

Open the src/app/contact.service.ts file and update it to look similar to the following code:

import { Injectable } from '@angular/core';

import { HttpClient } from “@angular/common/http”;

@Injectable({

providedIn: 'root'

})

export class ContactService {

API\_URL: string = "/api/";

constructor(private http: HttpClient) { }

  getContacts(): Observable<any[]> {

    return this.http.get<any[]>(this.API\_URL + 'contacts')

  }

  getContact(contactId: any){

    return this.http.get<any>(`${this.API\_URL + 'contacts'}/${contactId}`)

  }

}

We added two methods:

* getContacts()  
  For getting all contacts.
* getContact()  
  For getting a contact by id.

You can set the API\_URL to whatever URL since we are not going to use a real back-end. All requests will be intercepted and sent to the in-memory back-end.

### Creating Our Angular Components

Before we can see how to use the different Router features, let’s first create a bunch of components in our project.

Head over to your terminal and run the following commands:

$ ng g c contact-list

$ ng g c contact-detail

This will generate two ContactListComponent and ContactDetailComponent components and add them to the main app module.

### Setting Up Routing

Open the app-routing.module.ts file.

import { NgModule } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

We start by importing the NgModule from the @angular/core package which is a TypeScript decorator used to create an Angular module.

We also import the RouterModule and Routes classes from the @angular/router package . RouterModule provides static methods like RouterModule.forRoot() for passing a configuration object to the Router.

Next, we define a constant routes array of type Routes which will be used to hold information for each route.

Finally, we create and export a module called AppRoutingModule(You can call it whatever you want) which is simply a TypeScript class decorated with the @NgModule decorator that takes some meta information object. In the imports attribute of this object, we call the static RouterModule.forRoot(routes) method with the routes array as a parameter. In the exports array we add the RouterModule.

### IMPORTING THE ROUTING MODULE

Next, we need to import this module routing into the main app module that lives in the src/app/app.module.ts file:

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

We import the AppRoutingModule from ./app-routing.module and we add it in the imports array of the main module.

ADDING THE ROUTER OUTLET [#](https://www.smashingmagazine.com/2018/11/a-complete-guide-to-routing-in-angular/#adding-the-router-outlet)

Finally, we need to add the router outlet. Open the src/app/app.component.html file which contains the main app template and add the <router-outlet> component:

<router-outlet></router-outlet>

This is where the Angular Router will render the component that corresponds to current browser’s path.

That’s all steps we need to follow in order to manually setup routing inside an Angular project.

### CREATING ROUTES

Now, let’s add routes to our two components. Open the src/app/app-routing.module.ts file and add the following routes to the routes array:

const routes: Routes = [

{path: 'contacts' , component: ContactListComponent},

{path: 'contact/:id' , component: ContactDetailComponent}

];

Make sure to import the two components in the routing module:

import { ContactListComponent } from './components/contact-list/contact-list.component';

import { ContactDetailComponent } from './components/contact-detail/contact-detail.component';

Now we can access the two components from the /contacts and contact/:id paths.

### ADDING NAVIGATION LINKS

Next let’s add navigation links to our app template using the routerLink directive. Open the src/app/app.component.html and add the following code on top of the router outlet:

<h2><a [routerLink] = "'/contacts'">Contacts</a></h2>

Next we need to display the list of contacts in ContactListComponent. Open the src/app/contact-list.component.ts then add the following code:

import { Component, OnInit } from '@angular/core';

import { ContactService } from 'src/app/services/contact.service';

@Component({

selector: 'app-contact-list',

templateUrl: './contact-list.component.html',

styleUrls: ['./contact-list.component.css']

})

export class ContactListComponent implements OnInit {

contacts: any[] = [];

constructor(private contactService: ContactService) { }

ngOnInit() {

this.contactService.getContacts().subscribe((data : any[])=>{

console.log(data);

this.contacts = data;

})

}

}

We create a contacts array to hold the contacts. Next, we inject ContactService and we call the getContacts() method of the instance (on the ngOnInit life-cycle event) to get contacts and assign them to the contacts array.

Next open the src/app/components/contact-list/contact-list.component.html file and add:

<table style="width:100%">

<tr>

<th>Name</th>

<th>Email</th>

<th>Actions</th>

</tr>

<tr \*ngFor="let contact of contacts" >

<td>{{ contact.name }}</td>

<td>{{ contact.email }}</td>

<td>

<a [routerLink]="['/contact', contact.id]">Go to details</a>

</td>

</tr>

</table>

We loop through the contacts and display each contact’s name and email. We also create a link to each contact’s details component using the routerLink directive.

This is a screen shot of the component:



When we click on the *Go to details* link, it will take us to ContactDetailsComponent. The route has an id parameter, let’s see how we can access it from our component.

Open the src/app/components/contact-detail/contact-detail.component.ts file and change the code to look similar to the following code:

import { Component, OnInit } from '@angular/core';

import { ActivatedRoute } from '@angular/router';

import { ContactService } from 'src/app/services/contact.service';

@Component({

selector: 'app-contact-detail',

templateUrl: './contact-detail.component.html',

styleUrls: ['./contact-detail.component.css']

})

export class ContactDetailComponent implements OnInit {

contact: any;

constructor(private contactService: ContactService, private route: ActivatedRoute) { }

ngOnInit() {

this.route.paramMap.subscribe((params: any) => {

console.log(params.get('id'))

this.contactService.getContact(params.get('id'))

.subscribe((c: any) =>{

console.log(c);

this.contact = c;

})

});

}

}

We inject ContactService and ActivatedRoute into the component. In ngOnInit() life-cycle event we retrieve the *id* parameter that will be passed from the route and use it to get the contact’s details that we assign to a contact object.

Open the src/app/contact-detail/contact-detail.component.html file and add:

<h1> Contact # {{contact.id}}</h1>

<p>

Name: {{contact.name}}

</p>

<p>

Email: {{contact.email}}

</p>



When we first visit our application from 127.0.0.1:4200/, the outlet doesn’t render any component so let’s redirect the empty path to the contacts path by adding the following route to the routes array:

{path: '', pathMatch: 'full', redirectTo: 'contacts'}

We want to match the exact empty path, that’s why we specify the **full** match strategy.

# Angular – Forms

<https://www.tutorialspoint.com/angular8/angular8_forms.htm>

<https://www.simplilearn.com/tutorials/angular-tutorial/angular-forms>

Forms are used to handle user input data. Angular 8 supports two types of forms. They are **Template driven forms** and **Reactive forms**. This section explains about Angular 8 forms in detail.

## Template driven forms

Template driven forms is created using directives in the template. It is mainly used for creating a simple form application. Let’s understand how to create template driven forms in brief.

**Source code:** Angular Forms/forms-app

**Component**: form-component

## Create simple form

Let us create a sample application **(forms-app)** in Angular 8 to learn the template driven form.

Open command prompt and create new Angular application using below command –

ng new forms-app

## Configure Forms

Before understanding forms, let us learn how to configure forms in an application. To enable template driven forms, first we need to import **FormsModule** in **app.module.ts.** It is given below −

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

//import FormsModule here

import { FormsModule } from '@angular/forms';

imports: [

BrowserModule,

AppRoutingModule,

FormsModule //Assign FormsModule

],

Once, **FormsModule** is imported, the application will be ready for form programming.

Create a test component using Angular CLI as mentioned below −

ng generate component components/form

Add the below code in **form.component.html** file as follows −

<form #username="ngForm" (ngSubmit)="onClickSubmit(username.value)">

<input type="text" name="username" placeholder="username" ngModel>

<br/>

<br/>

<input type="submit" value="submit">

</form>

Here, we used **ngModel** attribute in **input** text field.

Create **onClickSubmit()** method inside **form.component.ts** file as shown below

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-form,

templateUrl: './form.component.html',

styleUrls: ['./form.component.scss']

})

export class FormComponent implements OnInit {

ngOnInit() {

}

onClickSubmit(result: any) {

console.log("You have entered : " + result.username);

}

}

Open app.component.html and change the content as specified below −

<app-form></app-form>

Finally, start your application (if not done already) using the below command −

ng serve

Now, run your application and you could see the form.

Enter **Peter** in input text field and enter submit. **onClickSubmit** function will be called and user entered text **Peter** will be send as an argument. **onClickSubmit** will print the user name in the console.

## Reactive Forms

**Reactive Forms** is created inside component class so it is also referred as model driven forms. Every form control will have an object in the component and this provides greater control and flexibility in the form programming. **Reactive Form** is based on structured data model. Let’s understand how to use Reactive forms in angular.

### Configure Reactive forms

To enable reactive forms, first we need to import **ReactiveFormsModule** in **app.module.ts** as follows:

:

import { ReactiveFormsModule } from '@angular/forms';

:

imports: [

:

ReactiveFormsModule //Assign here

]

### Create Reactive forms

Before moving to create Reactive forms, we need to understand about the following concepts,

* **FormControl** − Define basic functionality of individual form control
* **FormGroup** − Used to aggregate the values of collection form control
* **FormArray** − Used to aggregate the values of form control into an array
* **ControlValueAccessor** − Acts as an interface between Forms API to HTML DOM elements.

We need to import **FormGroup, FormControl** classes in FormComponent and add code as follows:

import { Component, OnInit } from '@angular/core';

import { FormGroup, FormControl } from '@angular/forms';

@Component({

selector: 'app-form,

templateUrl: './form.component.html',

styleUrls: ['./form.component.css']

})

export class FormComponent implements OnInit {

username;

formdata;

ngOnInit() {

this.formdata = new FormGroup({

username: new FormControl("Ajay Singala")

});

}

onClickSubmit(data) {this.username = data.username;}

}

Here,

* Created an instance of **formGroup** and set it to local variable, formdata.
* Crete an instance of **FormControl** and set it one of the entry in formdata.
* Created a **onClickSubmit()** method, which sets the local variable, **userName** with its argument.

Add the below code in **form.component.html** file.

<div>

<form [formGroup]="formdata" (ngSubmit)="onClickSubmit(formdata.value)" >

<input type=”text" name="username" placeholder="username"

formControlName = "username">

<br/>

<br/>

<input type="submit" value="Click here">

</form>

</div>

<p> Textbox result is: {{username}} </p>

Here,

* New form is created and set it’s **formGroup** property to formdata.
* New input text field is created and set is **formControlName to username.**
* **ngSubmit** event property is used in the form and set onClickSubmit() method as its value.
* **onClickSubmit()** method gets formdata values as its arguments.

Now, run your application and you will see the form. Enter a value in the text box, click on the “Click Here” button and the text will be echoed back.

## Forms Validation

**Source code:** Angular Forms/forms-app

**Component**: hero-form-component.ts, .html and .css, hero.ts, user.ts, src/styles.css,

Walkthrough the above files and explain.

# Angular – Deployment

<https://angular.io/guide/deployment>

<https://angular.io/cli/build>

<https://medium.com/coinmonks/how-to-deploy-an-angular-app-8db1af39f8c1>

<https://www.npmjs.com/package/angular-cli-ghpages>

**Repo**: <https://github.com/AjaySingala/forms-app.git>

**Source**: C:\Temp\ng-demos\forms-app

* Create a repository for your project.
* Add, commit and push your code to the repo.
* Add angular-cli-ghpages to your project.

ng add angular-cli-ghpages

* Deploy your project to GitHub pages with all default settings. Your project will be automatically built in production mode.

ng deploy --base-href=/<repositoryname>/

ng deploy --base-href https://AjaySingala.github.io/forms-app/

Which is the same as:

ng deploy <your-angular-project> --base-href=/<repositoryname>/

ng deploy forms-app --base-href https://AjaySingala.github.io/forms-app/

Please be aware of the --base-href option. It is necessary when your project will be deployed to a non-root folder.

* Your project should be available at https://<username>.github.io/<repositoryname>.

Navigate to: <https://AjaySingala.github.io/forms-app/>

## Minimal Todo Angular App Deployment to Azure using GitHub Actions and SonarCloud

### Local Sonar

* First, setup a project on SonarCloud.io and note the organization and project key values.
* Install sonar scanner

npm install sonar-scanner --save-dev

* Configure the SonarQube properties file **sonar-project.properties** on the Angular project

sonar.host.url=https://sonarcloud.io

sonar.projectKey=AjaySingala\_ngtodoapp

sonar.organization=june2019ajs

sonar.exclusions=\*\*/node\_modules/\*\*

* Integrate Angular project’s Karma code coverage with Sonarqube

Add a script called sonar to your **package.json**, You can give any key name!

"scripts": {

:  
 "sonar-scanner": "sonar-scanner"

:

}

* Finally, run the below command to integrate the Karma coverage with the Sonar server,

npm run sonars-canner

### Using GitHub Actions

<https://azureossd.github.io/2022/01/29/Angular-Deployment-on-App-Service-Linux/>

<https://sonarcloud.io/project/configuration?id=AjaySingala_ngtodoapp&analysisMode=GitHubManual>

<https://sonarcloud.io/project/configuration?id=AjaySingala_ngtodoapp&analysisMode=GitHubActions>

* Create the **SONAR\_TOKEN** secret in the gitHub repo.
* Add the following to the .yml file:

jobs:

  sonarcloud:

    name: SonarCloud

    runs-on: ubuntu-latest

    steps:

      - uses: actions/checkout@v2

        with:

          fetch-depth: 0  # Shallow clones should be disabled for a better relevancy of analysis

      - name: SonarCloud Scan

        uses: SonarSource/sonarcloud-github-action@master

        env:

          GITHUB\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}  # Needed to get PR information, if any

          SONAR\_TOKEN: ${{ secrets.SONAR\_TOKEN }}

* Configure the SonarQube properties file **sonar-project.properties** on the Angular project

sonar.host.url=https://sonarcloud.io

sonar.projectKey=AjaySingala\_ngtodoapp

sonar.organization=june2019ajs

sonar.exclusions=\*\*/node\_modules/\*\*

**Source**: C:\Temp\ngtodoapp\todo-app

**Repo**: <https://github.com/AjaySingala/ngtodoapp>

**App Service**: <https://ajs-todoapp-ng.azurewebsites.net>

**Sonar Project**: ngtodoapp

* Walk-thru C:\Temp\ngtodoapp\todo-app\.github\workflows\main\_ajs-todoapp-ng.yml
* Azure setting: AppService -> Settings -> Configuraiton -> General Settings -> Startup command:

pm2 serve /home/site/wwwroot --no-daemon –spa

Graphical user interface, application

Description automatically generated

## Adding Environments and Approval Steps

* GitHub repo -> Settings – Environments -> New Environment.
* Give a name to the env and add Required Users for manual approval.
* Save Protection Rules.
* Add steps to the .yml file:
  + Note the “**needs**” rule for **deploy-prod**.

  deploy-dev:

    runs-on: ubuntu-latest

    needs: build

    environment:

      name: 'dev'

      url: ${{ steps.deploy-to-webapp.outputs.webapp-url }}

    steps:

      - name: Download artifact from build job

        uses: actions/download-artifact@v2

        with:

          name: node-app

      # - name: unzip artifact for deployment

      #   run: unzip release.zip

      - name: 'Deploy to Azure Web App'

        id: deploy-to-webapp

        uses: azure/webapps-deploy@v2

        with:

          app-name: 'ajs-todoapp-ng'

          slot-name: 'Production'

          publish-profile: ${{ secrets.AZUREAPPSERVICE\_PUBLISHPROFILE\_8D0208CE443C4D9AB31FD17E995100EC }}

          package: .

  deploy-prod:

    runs-on: ubuntu-latest

    needs: [build, deploy-dev]

    environment:

      name: 'Production'

      url: ${{ steps.deploy-to-webapp.outputs.webapp-url }}

    steps:

      - name: Download artifact from build job

        uses: actions/download-artifact@v2

        with:

          name: node-app

      # - name: unzip artifact for deployment

      #   run: unzip release.zip

      - name: 'Deploy to Azure Web App'

        id: deploy-to-webapp

        uses: azure/webapps-deploy@v2

        with:

          app-name: 'ajs-todoapp-ng'

          slot-name: 'Production'

          publish-profile: ${{ secrets.AZUREAPPSERVICE\_PUBLISHPROFILE\_8D0208CE443C4D9AB31FD17E995100EC }}

          package: .

# Angular – Unit Testing

<https://www.browserstack.com/guide/how-to-perform-angular-unit-testing>

<https://angular.io/guide/testing>

## What is Angular Unit testing?

[Test-driven development](https://www.browserstack.com/guide/what-is-test-driven-development) (TDD) is the single most powerful tool you have for preventing bugs within your application. Angular Unit testing is the process of testing small and isolated pieces of code in your Angular application. This provides an added advantage to the users in the sense that they can add any new features without breaking any other part of their application.

Jasmine and Karma frameworks are used for Unit Testing of the Angular applications.

## Unit Testing with Jasmine and Karma for Angular Apps

**Jasmine** is a free and open-source [Behavior Driven Development (BDD)](https://www.browserstack.com/guide/tdd-vs-bdd-vs-atdd) framework. It can run on any JavaScript-enabled platform. It attempts to describe tests in a human-readable form so that even people other than the devs can understand the test case. Jasmine does not require DOM. This makes jasmine-core bear low overhead and no external dependencies.

Using Jasmine, one can perform test cases similar to user behavior on a website. It is very beneficial for [front-end testing](https://www.browserstack.com/guide/front-end-testing). It includes the [responsiveness testing](https://www.browserstack.com/responsive) of UI across various devices with different resolutions. One can also automate user behavior with custom delay and [wait time](https://www.browserstack.com/guide/wait-commands-in-selenium-webdriver) for simulating the actual user behavior. Jasmine has extensive community support and comprehensive documentation which makes it easy to use.

**Karma** is a task runner for our tests. It allows the users to execute their Jasmine test codes in multiple real-time browsers from the command line. This command-line also displays the result of the tests. It watches the files for changes and re-runs the tests automatically. By default, Angular runs on Karma.

## How to create an Angular test App?

There are certain prerequisites required before starting with the testing, the first being to write a basic test app.

In order to incorporate Jasmine and Karma, it is advised to use Angular CLI (Node.js) to create your Angular App. This also lets us create a simple Jasmine spec file named the same as the App file but ending in **.spec.ts.**

**Source: /Angular Demos/UnitTesting/simpleApp**

To create and build a new Angular app, use the following command in your Angular CLI:

ng new simpleApp

This command will create a new sample Angular App called **simpleApp**. Once the App is created, go to the parent directory of your app and run the below command to run your app in the browser.

cd simpleApp

ng serve

## How to write a unit test in Angular?

The Angular testing package includes two utilities called **TestBed** and **async**. **TestBed** is the main utility package. (Please see the**app.component.spec.ts** file below)

There are three main methods in this test file:

* **describe()** – It’s a suite of Test scripts that calls a global Jasmine function with two parameters: a string and a function. It also consists of **beforeEach** block.
* **it()** – It’s the smallest unit test case that is written to be executed, which calls a global Jasmine function with two parameters: a string and a function. Multiple **it()** statements can be written inside the **describe()**
* **expect()** – Every **it()** statement has a **expect()** function which takes a value and expects a return in true form

When the sample angular App is created, the test script file is also created alongside. It ends with **.spec.ts**. Below is what the initial test script file **app.component.spec.ts** looks like:

import { TestBed } from '@angular/core/testing';

import { RouterTestingModule } from '@angular/router/testing';

import { AppComponent } from './app.component';

describe('AppComponent', () => {

  beforeEach(async () => {

    await TestBed.configureTestingModule({

      imports: [

        RouterTestingModule

      ],

      declarations: [

        AppComponent

      ],

    }).compileComponents();

  });

  it('should create the app', () => {

    const fixture = TestBed.createComponent(AppComponent);

    const app = fixture.componentInstance;

    expect(app).toBeTruthy();

  });

  it(`should have as title 'simpleApp'`, () => {

    const fixture = TestBed.createComponent(AppComponent);

    const app = fixture.componentInstance;

    expect(app.title).toEqual('simpleApp');

  });

  it('should render title', () => {

    const fixture = TestBed.createComponent(AppComponent);

    fixture.detectChanges();

    const compiled = fixture.nativeElement as HTMLElement;

    expect(compiled.querySelector('.content span')?.textContent).toContain('simpleApp app is running!');

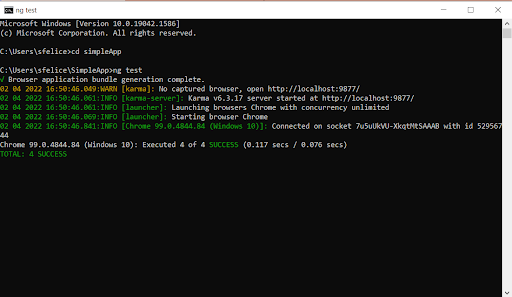
  });

});

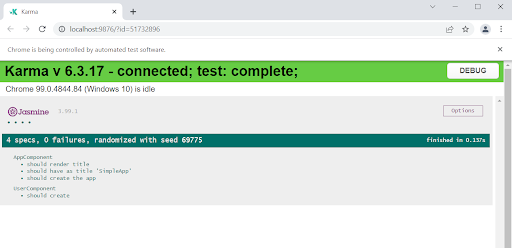
In the above code, there are three **it()** functions that equal three Unit test cases. Now, let’s run the above Unit test cases by using the following command in CLI:

ng test

The CLI will look like this:



A successful test will look something like this in the browser:



## How to write a negative unit test in Angular?

Now, let’s rewrite the above **app.component.spec.ts** file to showcase what a failed test will look like. Replace the code for the test case “should have as title 'simpleApp'” with the following code:

  it(`should have as title 'simpleApp'`, () => {

    const fixture = TestBed.createComponent(AppComponent);

    const app = fixture.componentInstance;

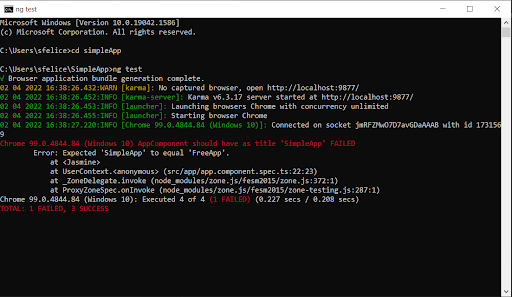
    // expect(app.title).toEqual('simpleApp');

    expect(app.title).toEqual('someOtherApp');

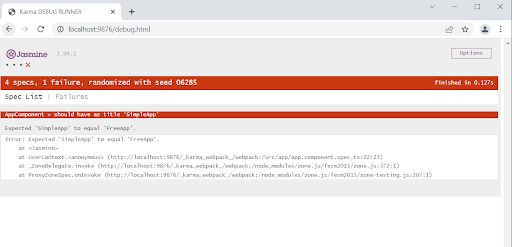
  });

The it() function has been modified to have the title as **someOtherApp**. This would result in a negative test since the title of our sample app is **SimpleApp**.

The CLI will look like this:



A failed test will look like this in the browser:



Similarly, multiple components can be created in your angular app, which will have their own test file that can be used to perform Unit testing.

# Angular – Unit Testing – TestBed

<https://angular.io/guide/testing-components-basics>

[TestBed](https://angular.io/api/core/testing/TestBed) is the primary api for writing unit tests for Angular applications and libraries.

Configures and initializes environment for unit testing and provides methods for creating components and services in unit tests.

<https://codecraft.tv/courses/angular/unit-testing/angular-test-bed/>

The Angular Test Bed (ATB) is a higher level Angular Only testing framework that allows us to easily test behaviours that depend on the Angular Framework.

In the beforeEach function for our test suite we configure a testing module using the TestBed class.

This creates a test Angular Module which we can use to instantiate components, perform dependency injection and so on.

## [Fixtures and DI](https://codecraft.tv/courses/angular/unit-testing/angular-test-bed/#_fixtures_and_di)

Once the ATB is setup we can then use it to instantiate components and resolve dependencies, like so:

import {TestBed, ComponentFixture} from '@angular/core/testing';

import {LoginComponent} from './login.component';

import {AuthService} from "./auth.service";

describe('Component: Login', () => {

let component: LoginComponent;

let fixture: ComponentFixture<LoginComponent>; (1)

let authService: AuthService;

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [LoginComponent],

providers: [AuthService]

});

// create component and test fixture

fixture = TestBed.createComponent(LoginComponent); (2)

// get test component from the fixture

component = fixture.componentInstance; (3)

// UserService provided to the TestBed

authService = TestBed.get(AuthService); (4)

});

});

1. A fixture is a wrapper for a component and its template.
2. We create an instance of a component fixture through the TestBed, this injects the AuthService into the component constructor.
3. We can find the actual component from the componentInstance on the fixture.
4. We can get resolve dependencies using the TestBed injector by using the get function.

## [Test Specs](https://codecraft.tv/courses/angular/unit-testing/angular-test-bed/#_test_specs)

Now we’ve configured the TestBed and extracted the component and service we can run through the same test specs as before:

it('needsLogin returns true when the user has not been authenticated', () => {

spyOn(authService, 'isAuthenticated').and.returnValue(false);

expect(component.needsLogin()).toBeTruthy();

expect(authService.isAuthenticated).toHaveBeenCalled();

});

it('needsLogin returns false when the user has been authenticated', () => {

spyOn(authService, 'isAuthenticated').and.returnValue(true);

expect(component.needsLogin()).toBeFalsy();

expect(authService.isAuthenticated).toHaveBeenCalled();

});

## [When to Use ATB](https://codecraft.tv/courses/angular/unit-testing/angular-test-bed/#_when_to_use_atb)

* It allows us to test the interaction of a directive or component with its template.
* It allows us to easily test change detection.
* It allows us to test and use Angular’s DI framework.
* It allows us to test using the NgModule configuration we use in our application.
* It allows us to test user interaction via clicks and input fields

## Component class testing

<https://angular.io/guide/testing-components-basics>

**Source: /Angular Demos/UnitTesting/simpleApp**

Test a component class on its own as you would test a service class.

Component class testing should be kept very clean and simple. It should test only a single unit. At first glance, you should be able to understand what the test is testing.

Consider this LightSwitchComponent which toggles a light on and off (represented by an on-screen message) when the user clicks the button.

ng g c components/LightSwitch

// light-switch.component.ts

import { Component, OnInit } from '@angular/core';

@Component({

  selector: 'app-light-switch',

  templateUrl: './light-switch.component.html',

  styleUrls: ['./light-switch.component.css']

})

export class LightSwitchComponent implements OnInit {

  isOn = false;

  clicked() {

    this.isOn = !this.isOn;

  }

  get message() {

    return `The light is ${this.isOn ? 'On' : 'Off'}`;

  }

  constructor() { }

  ngOnInit(): void {

  }

}

You might decide only to test that the clicked() method toggles the light's on/off state and sets the message appropriately.

This component class has no dependencies. To test these types of classes, follow the same steps as you would for a service that has no dependencies:

1. Create a component using the new keyword.
2. Poke at its API.
3. Assert expectations on its public state.

// light-switch.component.spec

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { LightSwitchComponent } from './light-switch.component';

describe('LightSwitchComponent', () => {

  let component: LightSwitchComponent;

  let fixture: ComponentFixture<LightSwitchComponent>;

  beforeEach(async () => {

    await TestBed.configureTestingModule({

      declarations: [ LightSwitchComponent ]

    })

    .compileComponents();

    fixture = TestBed.createComponent(LightSwitchComponent);

    component = fixture.componentInstance;

    fixture.detectChanges();

  });

  it('should create', () => {

    expect(component).toBeTruthy();

  });

  it('#clicked() should toggle #isOn', () => {

    const comp = new LightSwitchComponent();

    expect(comp.isOn)

      .withContext('off at first')

      .toBe(false);

    comp.clicked();

    expect(comp.isOn)

      .withContext('on after click')

      .toBe(true);

    comp.clicked();

    expect(comp.isOn)

      .withContext('off after second click')

      .toBe(false);

  });

  it('#clicked() should set #message to "is on"', () => {

    const comp = new LightSwitchComponent();

    expect(comp.message)

      .withContext('off at first')

      .toMatch(/is off/i);

    comp.clicked();

    expect(comp.message)

      .withContext('on after clicked')

      .toMatch(/is on/i);

  });

});

## Component and Associated Class Testing

Now, let’s create and test the DashboardHeroComponent.

ng g class classes/hero

ng g c components/DashboardHero

// hero.ts

export class Hero {

    constructor(

        id: number,

        name: string

    ) {}

}

// dashboard-hero.component.ts

import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

import { Hero } from 'src/app/classes/hero';

@Component({

  selector: 'app-dashboard-hero',

  templateUrl: './dashboard-hero.component.html',

  styleUrls: ['./dashboard-hero.component.css']

})

export class DashboardHeroComponent implements OnInit {

  @Input() hero!: Hero;

  @Output() selected = new EventEmitter<Hero>();

  click() {

    this.selected.emit(this.hero);

  }

  constructor() { }

  ngOnInit(): void {

  }

}

It appears within the template of a parent component, which binds a hero to the @[Input](https://angular.io/api/core/Input) property and listens for an event raised through the selected @[Output](https://angular.io/api/core/Output) property.

You can test that the class code works without creating the DashboardHeroComponent or its parent component.

// dashboard-hero.component.spec

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { first } from 'rxjs';

import { Hero } from 'src/app/classes/hero';

import { DashboardHeroComponent } from './dashboard-hero.component';

describe('DashboardHeroComponent', () => {

  let component: DashboardHeroComponent;

  let fixture: ComponentFixture<DashboardHeroComponent>;

  beforeEach(async () => {

    await TestBed.configureTestingModule({

      declarations: [ DashboardHeroComponent ]

    })

    .compileComponents();

    fixture = TestBed.createComponent(DashboardHeroComponent);

    component = fixture.componentInstance;

    fixture.detectChanges();

  });

  it('should create', () => {

    expect(component).toBeTruthy();

  });

  it('raises the selected event when clicked', () => {

    const comp = new DashboardHeroComponent();

    const hero: Hero = {id: 42, name: 'Test'};

    comp.hero = hero;

    comp.selected.pipe(first()).subscribe((selectedHero: Hero) => expect(selectedHero).toBe(hero));

    comp.click();

  });

});

## Testing Components with Dependencies (mocking a service)

When a component has dependencies, you might want to use the [TestBed](https://angular.io/api/core/testing/TestBed) to both create the component and its dependencies.

The following WelcomeComponent depends on the UserService to know the name of the user to greet.

ng g service services/user

ng g c components/welcome

ng g class services/Mock/MockUserService

// user.service.ts

import { Injectable } from '@angular/core';

@Injectable({

  providedIn: 'root'

})

export class UserService {

  isLoggedIn = true;

  user = {name: 'Ajay Singala'};

  constructor() { }

}

// welcome.component.ts

import { Component, OnInit } from '@angular/core';

import { UserService } from 'src/app/services/user.service';

@Component({

  selector: 'app-welcome',

  templateUrl: './welcome.component.html',

  styleUrls: ['./welcome.component.css']

})

export class WelcomeComponent implements OnInit {

  welcome = '';

  constructor(private userService: UserService) { }

  ngOnInit(): void {

    this.welcome = this.userService.isLoggedIn ? 'Welcome, ' + this.userService.user.name : 'Please log in.';

  }

}

You might start by creating a mock of the UserService that meets the minimum needs of this component.

// mock-user-service.ts

export class MockUserService {

    isLoggedIn = true;

    user = { name: 'Test User'};

}

Then provide and inject both the **component** and the service in the [TestBed](https://angular.io/api/core/testing/TestBed) configuration.

// welcome.component.spec

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { MockUserService } from 'src/app/services/Mock/mock-user-service';

import { UserService } from 'src/app/services/user.service';

import { WelcomeComponent } from './welcome.component';

describe('WelcomeComponent', () => {

  let component: WelcomeComponent;

  let fixture: ComponentFixture<WelcomeComponent>;

  let comp: WelcomeComponent;

  let userService: UserService;

  beforeEach(async () => {

    await TestBed.configureTestingModule({

      // provide the component-under-test and dependent service.

      providers: [

        WelcomeComponent,

        { provide: UserService, useClass: MockUserService }

      ],

      declarations: [ WelcomeComponent ]

    })

    .compileComponents();

    // inject both the component and the dependent service.

    comp = TestBed.inject(WelcomeComponent);

    userService = TestBed.inject(UserService);

    fixture = TestBed.createComponent(WelcomeComponent);

    component = fixture.componentInstance;

    fixture.detectChanges();

  });

  it('should create', () => {

    expect(component).toBeTruthy();

  });

  it('should not have welcome message after construction', () => {

    expect(comp.welcome).toBe('');

  });

  it('should welcome logged in user after Angular calls ngOnInit', () => {

    comp.ngOnInit();

    expect(comp.welcome).toContain(userService.user.name);

  });

  it('should ask user to log in if not logged in after ngOnInit', () => {

    userService.isLoggedIn = false;

    comp.ngOnInit();

    expect(comp.welcome).not.toContain(userService.user.name);

    expect(comp.welcome).toContain('log in');

  });

});

## Mocking

<https://codecraft.tv/courses/angular/unit-testing/mocks-and-spies/>

**Source: /Angular Demos/UnitTesting/simpleApp**

Let’s imagine we have a LoginComponent which works with the AuthService we tested in the previous lecture, like so:

ng g class services/AuthService

ng g component components/login

// AuthService.ts

export class AuthService {

    isAuthenticated(): boolean {

        return !!localStorage.getItem('token');

    }

}

// login.component.ts

import { Component, OnInit } from '@angular/core';

import { AuthService } from 'src/app/services/auth-service';

@Component({

  selector: 'app-login',

  template: `<a [hidden]="needsLogin()">Login</a>`

})

export class LoginComponent implements OnInit {

  constructor(private auth: AuthService) { }

  ngOnInit(): void {

  }

  needsLogin() {

    return !this.auth.isAuthenticated();

  }

}

We inject the AuthService into the LoginComponent and the component shows a Login button if the AuthService says the user isn’t authenticated.

### [Testing with the Real AuthService](https://codecraft.tv/courses/angular/unit-testing/mocks-and-spies/#_testing_with_the_real_authservice)

We could test the LoginComponent by using a real instance of AuthService but if you remember to trick AuthService into returning true for the isAuthenticated function we needed to setup some data via localStorage.

// login.component.spec.ts

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { AuthService } from 'src/app/services/auth-service';

import { LoginComponent } from './login.component';

describe('LoginComponent', () => {

  let component: LoginComponent;

  let service: AuthService;

  beforeEach(async () => {  // (1)

    service = new AuthService();

    component = new LoginComponent(service);

  });

  afterEach(() => {       // (2)

    localStorage.removeItem('token');

  });

  it('needsLogin returns true when the user has not been authenticated', () => {

    expect(component.needsLogin()).toBeTruthy();

  });

  it('needsLogin returns false when the user has been authenticated', () => {

    localStorage.setItem('token', '12345');   // (3)

    expect(component.needsLogin()).toBeFalsy();

  });

});

* 1. We create an instance of AuthService and inject it into out LoginComponent when we create it.
  2. We clean up data and localStorage after each test spec has been run.
  3. We setup some data in localStorage in order to get the behaviour we want from AuthService.

So in order to test LoginComponent we would need to know the inner workings of AuthService.

That’s not very isolated but also not too much to ask for in this scenario. However imagine if LoginComponent required a number of other dependencies in order to run, we would need to know the inner workings of a number of other classes just to test our LoginComponent.

This results in Tight Coupling and our tests being very Brittle, i.e. likely to break easily. For example if the AuthService changed how it stored the token, from localStorage to cookies then the LoginComponent test would break since it would still be setting the token via localStorage.

This is why we need to test classes in isolation, we just want to worry about LoginComponent and not about the myriad of other things LoginComponent depends on.

We achieve this by Mocking our dependencies. Mocking is the act of creating something that looks like the dependency but is something we control in our test. There are a few methods we can use to create mocks.

### M[ocking with Fake Classes](https://codecraft.tv/courses/angular/unit-testing/mocks-and-spies/#_mocking_with_fake_classes)

We can create a fake AuthService called MockedAuthService which just returns whatever we want for our test.

We can even remove the AuthService import if we want, there really is no dependency on anything else. The LoginComponent is tested in isolation:

Copy login.component.spec.ts to login-mock.component.spec.ts.

// login-mock.component.spec.ts

import { LoginComponent } from './login.component';

class MockAuthService {   // (1)

  authenticated = false;

  isAuthenticated() {

    return this.authenticated;

  }

}

describe('LoginComponent', () => {

  let component: LoginComponent;

  let service: MockAuthService;

  beforeEach(async () => {  // (2)

    service = new MockAuthService();

    component = new LoginComponent(service);

  });

  afterEach(() => {

    // service = null;

    // component = null;

  });

  it('needsLogin returns true when the user has not been authenticated', () => {

    service.authenticated = false;  // (3)

    expect(component.needsLogin()).toBeTruthy();

  });

  it('needsLogin returns false when the user has been authenticated', () => {

    service.authenticated = true;   // (3)

    expect(component.needsLogin()).toBeFalsy();

  });

});

1. We create a class called MockAuthService which has the same isAuthenticated function as the real AuthService class. The one difference is that we can control what isAuthenticated returns by setting the value of the authenticated property.
2. We inject into our LoginComponent an instance of the MockAuthService instead of the real AuthService.
3. In our tests we trigger the behaviour we want from the service by setting the authenticated property.

By using a fake MockAuthService we:

* Don’t depend on the real AuthService, in fact we don’t even need to import it into our specs.
* Make our code less brittle, if the inner workings of the real AuthService ever changes our tests will still be valid and still work.

### [Mocking by Overriding Functions](https://codecraft.tv/courses/angular/unit-testing/mocks-and-spies/#_mocking_by_overriding_functions)

Sometimes creating a complete fake copy of a real class can be complicated, time consuming and unnecessary.

We can instead simply extend the class and override one or more specific function in order to get them to return the test responses we need, like so:

class MockAuthService extends AuthService {

authenticated = false;

isAuthenticated() {

return this.authenticated;

}

}

In the above class MockAuthService extends AuthService. It would have access to all the other functions and properties that exist on AuthService but only override the isAuthenticated function so we can easily control it’s behaviour and isolate our LoginComponent test.

### [Mocking with Spies](https://codecraft.tv/courses/angular/unit-testing/mocks-and-spies/#_mocking_with_spies)

A Spy is a feature of Jasmine which lets you take an existing class, function, or object and mock it in such a way that you can control what gets returned from function calls.

Let’s re-write our test to use a Spy on a real instance of AuthService instead, like so:

Copy login.component.spec.ts to login-spy.component.spec.ts.

// login-spy.component.spec.ts

import { AuthService } from 'src/app/services/auth-service';

import { LoginComponent } from './login.component';

describe('LoginComponent', () => {

  let component: LoginComponent;

  let service: AuthService;

  let spy: any;

  beforeEach(async () => {  // (1)

    service = new AuthService();

    component = new LoginComponent(service);

  });

  afterEach(() => {       // (2)

    localStorage.removeItem('token');

  });

  it('needsLogin returns true when the user has not been authenticated', () => {

    spy = spyOn(service, 'isAuthenticated').and.returnValue(false); // (3)

    expect(component.needsLogin()).toBeTruthy();

    expect(service.isAuthenticated).toHaveBeenCalled(); // (4)

  });

  it('needsLogin returns false when the user has been authenticated', () => {

    spy = spyOn(service, 'isAuthenticated').and.returnValue(true);

    expect(component.needsLogin()).toBeFalsy();

    expect(service.isAuthenticated).toHaveBeenCalled();

  });

});

1. We create a real instance of AuthService and inject it into the LoginComponent.
2. In our teardown function there is no need to delete the token from localStorage.
3. We create a *spy* on our service so that if the isAuthenticated function is called it returns false.
4. We can even check to see if the isAuthenticated function was called.

By using the spy feature of Jasmine we can make any function return anything we want:

spyOn(service, 'isAuthenticated').and.returnValue(false);

In our example above we make the isAuthenticated function return false or true in each test spec according to our needs.

# Angular – Unit Testing – Code Coverage

<https://angular.io/guide/testing-code-coverage>

The CLI can run unit tests and create code coverage reports. Code coverage reports show you any parts of your code base that might not be properly tested by your unit tests.

To generate a coverage report run the following command in the root of your project.

ng test --no-watch --code-coverage

OR

ng test --code-coverage

When the tests are complete, the command creates a new /coverage folder in the project. Open the index.html file to see a report with your source code and code coverage values.

If you want to create code-coverage reports every time you test, set the following option in the CLI configuration file, angular.json:

content\_copy"test": {

"options": {

"codeCoverage": true

}

}