
Table of Contents

| | |
|----------------------------|-----|
| Introduction | 1.1 |
| Setting up the environment | 1.2 |
| Angular CLI | 1.3 |
| References | 1.4 |

TypeScript

| | |
|-------------------------|-----|
| TypeScript Fundamentals | 2.1 |
|-------------------------|-----|

Angular Application Structure

| | |
|------------------|-----|
| Angular Features | 3.1 |
| LifeCycle | 3.2 |

Components

| | |
|-------------------|-----|
| Components | 4.1 |
| Working with data | 4.2 |
| Event binding | 4.3 |

Directives

| | |
|-----------------------|-----|
| Directives | 5.1 |
| Structural Directives | 5.2 |
| Attribute Directives | 5.3 |
| Custom Directives | 5.4 |

Working with multiple components

| | |
|------------------|-----|
| Input properties | 6.1 |
|------------------|-----|

| | |
|-------------------|-----|
| Output properties | 6.2 |
|-------------------|-----|

Pipes

| | |
|----------------|-----|
| Introduction | 7.1 |
| Built In Pipes | 7.2 |
| Custom Pipes | 7.3 |

Services

| | |
|----------------------|-----|
| Services | 8.1 |
| Dependency Injection | 8.2 |
| http | 8.3 |

Forms

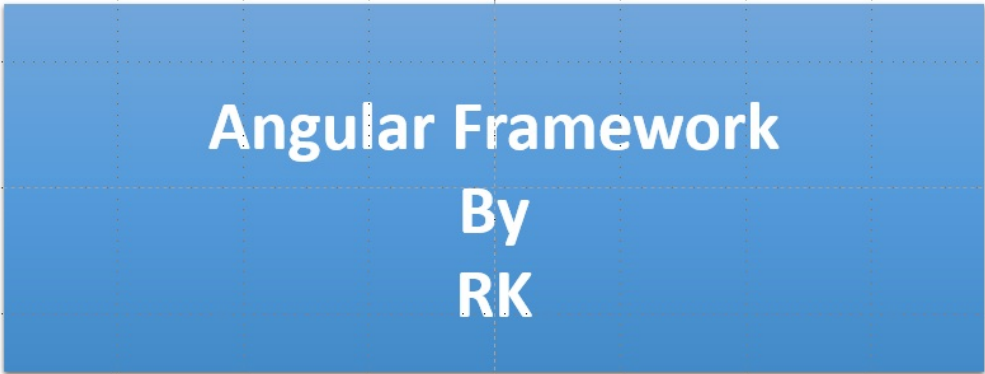
| | |
|-----------------------|-----|
| Forms | 9.1 |
| Validations | 9.2 |
| Reactive Forms | 9.3 |
| Template driven forms | 9.4 |

Routing

| | |
|---------------------|------|
| Routing | 10.1 |
| Implementing Routes | 10.2 |
| Route Params | 10.3 |
| Router | 10.4 |
| Guards | 10.5 |

Examples

| | |
|---------|------|
| 1 basic | 11.1 |
|---------|------|



Angular Framework

By
RK

mail-id: javabyrk@gmail.com.

we are going to use Node.js and npm (node package manager) for tooling purposes. We need

them for downloading tools, libraries, and packages.

- Node.js is a platform built on top of V8, Google's JavaScript runtime, which also powers the Chrome browser
- Node.js is used for developing server-side JavaScript applications
- The npm is a package manager for Node.js, which makes it quite simple to install additional tools via packages; it comes bundled with Node.js

Installing Node.js and npm

Download the Node.js from <https://nodejs.org/en/download/> . Download the latest version of Node for the respective operating system and install it. The npm is installed as part of the Node.js installation.

Once we install Node.js, run the following commands on the command line in Windows or Terminal in macOS to verify that Node.js and npm are installed and set up properly:

```
$ node -v
```

```
$ npm -v
```

```
.
```

To write a simple hello world application, we initially had to create a lot of files with boilerplate code and project configuration. This process is common for both small and large applications.

For large applications, we create a lot of modules, components, services, directives, and pipes with boilerplate code and project configuration. This is a very time-consuming process. Since we want to save time and be productive by focusing on solving business problems instead of spending time on tedious tasks, tooling comes in handy.

The Angular team created a command-line tool know as Angular CLI. The Angular CLI helps us in generating Angular projects with required configurations, boilerplate code, and also downloads the required node packages with one simple command. It also provides commands for generating components, directives, pipes, services, classes, guards, interfaces, enums, modules, modules with routing and building, running and testing the applications locally.

Getting started with Angular CLI

The Angular CLI is available as a node package. First, we need to download and install it with the following command:

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

The preceding command will install Angular CLI, we can then access it anywhere via command line or Terminal.

Working with First Example

To generate the Angular project using CLI we can use the `ng g new project-name` command.

```
ng new hello-world
```

This command creates a folder named hello-world, generates Angular project under it with all the required files and downloads all the node packages.

Once you run it, you'll see the following output:

```
installing ng2
2 create .editorconfig
3 create README.md
4 create src/app/app.component.css
5 create src/app/app.component.html
6 create src/app/app.component.spec.ts
7 create src/app/app.component.ts
8 create src/app/app.module.ts
9 create src/assets/.gitkeep
10 create src/environments/environment.prod.ts
11 create src/environments/environment.ts
12 create src/favicon.ico
13 create src/index.html
14 create src/main.ts
15 create src/polyfills.ts
16 create src/styles.css
17 create src/test.ts
18 create src/tsconfig.json
19 create .angular-cli.json
20 create e2e/app.e2e-spec.ts
21 create e2e/app.po.ts
22 create e2e/tsconfig.json
23 create .gitignore
24 create karma.conf.js
25 create package.json
26 create protractor.conf.js
27 create tslint.json
28 Successfully initialized git.
29 Installing packages for tooling via npm.
30 Installed packages for tooling via npm.
```

To run the application, we need to navigate to the project folder and run the `ng serve` command:

```
$ cd hello-world
$ ng serve
```

The `ng serve` command compiles and builds the project, and starts the local web server at <http://localhost:4200> URL. When we navigate to <http://localhost:4200> URL in the browser, we see the following output:



app works!

Integrating with other frameworks.

materialize

for reference we use the following url.

<https://material.angular.io/>

1 . create the angular project by using the ng new command.

```
ng new example1
```

2 . change the directory to example1.

```
cd example1
```

3 . install the materialize dependencies using the node package manager.

```
npm install --save @angular/material @angular/cdk
```

4 . import the materialize modules into the main module of our project(app module).

```
import {MatButtonModule, MatCheckboxModule} from '@angular/material';

@NgModule({
  ...
  imports: [MatButtonModule, MatCheckboxModule],
  ...
})
export class AppModule { }
```

BootStrap

for reference we use the following url.

<https://ng-bootstrap.github.io/#/getting-started>

1 . create the angular project by using the ng new command.

```
ng new example1
```

2 . change the directory to example1.

```
cd example1
```

3 . install the bootstrap dependencies using the node package manager.

```
npm install --save @ng-bootstrap/ng-bootstrap
```

4 . import the bootstrap modules into the main module of our project(app module).

```
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

@NgModule({
  declarations: [AppComponent, ...],
  imports: [NgbModule.forRoot(), ...],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

TypeScript is a superset of JavaScript, which means all the code written in JavaScript is valid TypeScript code, and TypeScript compiles back to simple standards-based JavaScript code, which runs on any browser, for any host, on any OS.

```
All the ECMAScript 5 code is valid in js6. all js6 code is valid TypeScript code.
```

Installing TypeScript

Once you have Node.js setup, the next step is to install TypeScript. Make sure you install at least version 2.1 or greater.

To install it, run the following npm command:

```
> npm install -g typescript
```

The preceding command will install the TypeScript compiler and makes it available globally.

We can compile TypeScript code into JavaScript by invoking the TypeScript compiler using the following command:

```
> tsc <filename.ts>
```

DataTypes

Important data types in TypeScript.

String: -

In TypeScript, we can use either double quotes (") or single quotes (') to surround strings similar to JavaScript.

```
var bookName: string = "Angular";
```

Number: -

As in JavaScript, all numbers in TypeScript are floating point values:

```
var version: number = 4;
```

Boolean: -

The boolean data type represents the true/false value:

```
var isCompleted: boolean = false;
```

Array: -

We have two different syntaxes to describe arrays, and the first syntax uses the element type followed by []:

```
var frameworks: string[] = ['Angular', 'React', 'Ember'];
```

The second syntax uses a generic array type, `Array<elementType>`:

```
var frameworks: Array<string> = ['Angular', 'React', 'Ember'];
```

Any: -

If we need to opt out type-checking in TypeScript to store any value in a variable whose type is not known right away, we can use any keyword to declare that variable:

```
var eventId: any = 7890;
```

```
eventId = 'event1';
```

Void: -

The void keyword represents not having any data type. Functions without return keyword do not return any value, and we use void to represent it.

```
function simpleMessage(): void {
```

```
    alert\("Hey! I return void"\);
```

```
}
```

Functions: -

Functions are the fundamental building blocks of any JavaScript application. In JavaScript, we declare functions in two ways.

Function declaration – named function

The following is an example of function declaration:

```
function sum(a, b) {  
    return a + b;  
}
```

Function expression – anonymous function

The following is an example of function expression:

```
var result = function(a, b) {  
    return a + b;  
}
```

Classes: -

In JavaScript ES5 object oriented programming was accomplished by using prototype-based objects.

```
var Person = (function () {  
    function Person(name) {  
        this.name = name;  
    }  
    Person.prototype.sayHello = function () {  
        return 'Hello ' + this.name;  
    };  
    return Person;  
})();  
var person = new Person('Shravan');  
console.log(person.name);  
console.log(person.sayHello());
```

in ES6 we finally have built-in classes in JavaScript.

To define a class we use the new class keyword and give our class a name and a body:

```
class Person{  
  
    name;  
  
    sayHello(){  
        return 'Hello'+this.name;  
    }  
  
}
```

Classes may have properties, methods, and constructors.

Properties: -

Properties define data attached to an instance of a class. For example, a class named `Person` might have properties like `first_name`, `last_name` and `age`.

Each property in a class can optionally have a type. For example, we could say that the `first_name` and `last_name` properties are strings and the `age` property is a number.

The declaration for a `Person` class that looks like this:

```
class Person {  
    first_name: string;  
    last_name: string;  
    age: number;  
}
```

Methods: -

Methods are functions that run in context of an object. To call a method on an object, we first have to have an instance of that object.

To instantiate a class, we use the `new` keyword. Use `new Person()` to create a new instance of the `Person` class.

if we want to create the method in a class we need to do the following way.

```
class Person {  
    first_name: string;  
    last_name: string;  
    age: number;  
  
    getFullName(){  
        return this.first_name+' '+this.last_name;  
    }  
}
```

// declare a variable of type `Person`

`var p: Person;`

// instantiate a new `Person` instance

`p = new Person();`

```
// give it a first_name  
p.first_name = 'Raju';  
  
// call the getFullName method  
p.getFullName();
```

Constructors: -

A constructor is a special method that is executed when a new instance of the class is being created.

Constructor methods must be named constructor. They can optionally take parameters but they can't return any values, since they are called when the class is being instantiated (i.e. an instance of the class is being created, no other value can be returned).

```
class Person {  
  
    firstName = "";  
    lastName = "";  
  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    getFullName(){  
        return this.firstName+' '+this.lastName;  
    }  
  
}
```

Inheritance: -

Inheritance is a way to indicate that a class receives behavior from a parent class. Then we can override, modify or augment those behaviors on the new class.

Person class:

```
class Person {  
  
    firstName = "";  
    lastName = "";  
  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    getFullName(){  
        return this.firstName+' '+this.lastName;  
    }  
  
}
```

Student class: -

```
class Student extends Person {  
    course;  
  
    constructor(firstName, lastName, course) {  
        super(firstName, lastName);  
        this.course = course;  
    }  
  
    getDetails() {  
        return `${super.getFullName()} and i'm studying ${this.course}`;  
    }  
}
```

Interfaces: -

Interfaces provides the structure for the data.

```
interface Human {  
    firstName: string;  
    lastName: string;  
    getFullName?: Function;  
}
```

Default parameters and values

One of the characteristics of JavaScript is that it allows developers to call a function with any number of arguments:

- if you pass more arguments than the number of the parameters, the extra arguments are ignored (well, you can still use them with the special arguments variable, to be accurate).
- if you pass less arguments than the number of the parameters, the missing parameter will be set to undefined.

```
function m1(size = 10, page = 1) {  
    // ...  
}
```

here the size and page variables default values are 10 and 1.

Variable hoisting

a variable which declares at the top of the function, even if you declared it later. we have only two scopes in the JS. function scope and global scope. we don't have block scope. to solve this problem we use latest variable creation syntax by using let.

let has been introduced to replace var in the long run, so you can pretty much drop the good old var keyword and start using let instead.

Constants

ES6 introduces const to declare... constants! When you declare a variable with const, it has to be initialized and you can't assign another value later.

```
const DEPLOYMENT_MODE = 'dev';
```

As for variables declared with let, constants are not hoisted and are only declared at the block level.

```
const student = {};  
student.id = '1'; // valid assignment.
```

Arrow functions

One very useful feature in ES6 is the new arrow function syntax, using the 'fat arrow' operator (\Rightarrow). It is SO useful for callbacks and anonymous functions!

```
let add1 = function(a,b){  
  return a+b;  
}
```

we can right the same code with much simpler syntax.

```
let add2 = (a,b) => a+b;
```

Another example for the same is

```
getUser(login)  
  .then(function (user) {  
    return getRights(user); // getRights is returning a promise  
  })  
  .then(function (rights) {  
    return updateMenu(rights);  
  })
```

with arrow functions

```
getUser(login)  
  .then(user => getRights(user))  
  .then(rights => updateMenu(rights))
```

Arrows are a great way to cleanup your inline functions. It makes it even easier to use higher-order functions in JavaScript.

Template Strings

In ES6 new template strings were introduced. The two great features of template strings are

1. Variables within strings (without being forced to concatenate with +) and
2. Multi-line strings

Variables in strings

The idea is that you can put variables right in your strings. means we can inject the values into the string.

```
var firstName = "Nate";
var lastName = "Murray";

// interpolate a string
var greeting = `Hello ${firstName} ${lastName}`;

console.log(greeting);
```

you must enclose your string in backticks not single or double quotes.

Multiline strings

Another great feature of backtick strings is multi-line strings:

```
var template = `
    <div>
        <h1>Hello</h1>
        <p>This is a great website</p>
    </div>
`;
```

Multiline strings are a huge help when we want to put strings in our code that are a little long, like templates.

Sets and Maps

we have new collections in the JS6.

To represent key and value pairs we use the maps.

```
let map = new Map();
map.set("A", 1);
map.set("B", 2);
map.set("C", 3);
```

Map API methods are

- 1) set
- 2) get
- 3) has
- 4) delete

5) clear

```
for (let [key, value] of map) {  
  console.log(key, value);  
}  
console.log(map.get("A"));  
console.log(map.has("A"));  
console.log(map.size);  
  
map.delete("A");  
console.log(map.size);  
  
map.clear();  
console.log(map.size);
```

Set also represents the group of values like array. but it wont allow the duplicate values.

```
// Set
let set = new Set();
set.add('A');
set.add('B');
set.add('C');

let set2 = new Set()
  .add('A')
  .add('B')
  .add('C');

let set3 = new Set(['A', 'B', 'C']);

console.log(set.has('A'));

set.delete('A');

console.log(set.size);

set.clear();
console.log(set.size);

let set4 = new Set();
set3.add('B');
console.log(set3.size);
// 1
set4.add('B');
console.log(set4.size);
// 1

for (let entry of set2) {
  console.log(entry);
}
```

Modules

A standard way to organize functions in namespaces and to dynamically load code in JS has always

been lacking. NodeJS has been one of the leaders in this. JS6 aims to create a syntax using the best from both worlds, without caring about the actual

implementation. The new syntax handles

how you export and import things to and from modules.

In `student_services.js`:

```
export function create(a,b) {  
  // ...  
}  
export function update(c) {  
  // ...  
}
```

the new keyword `export` does a straightforward job and exports the two functions.

In other js files we can simply define the following way

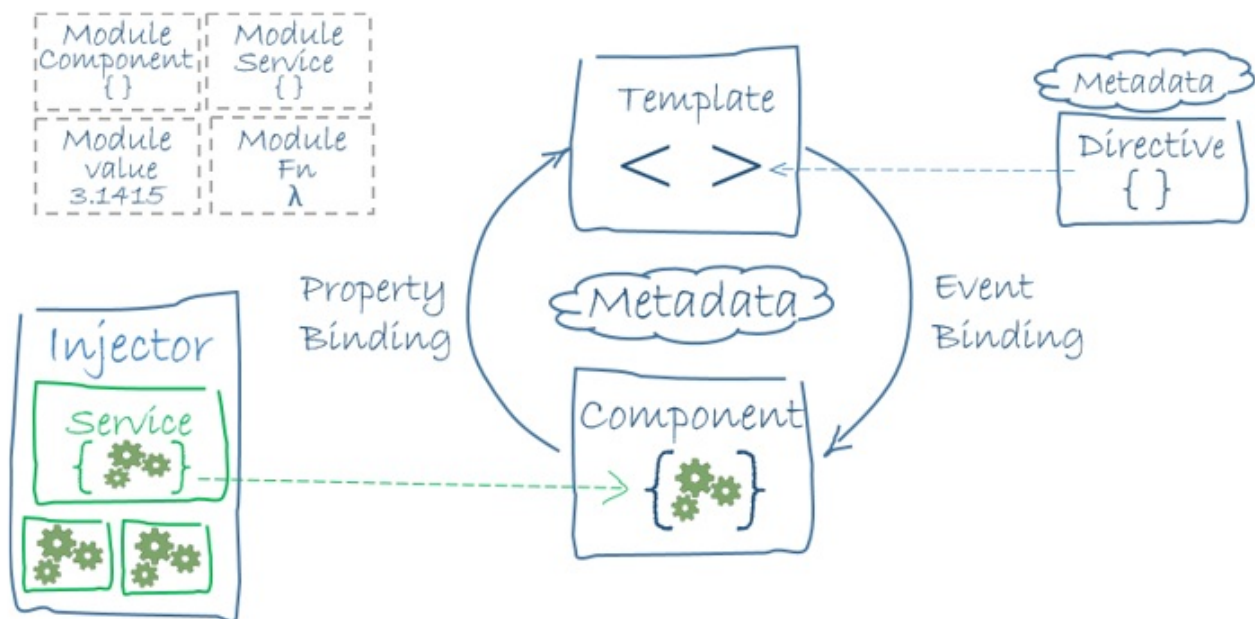
```
import { create, update } from './student_services';
```

```
// later in the file  
create(10, "RK");  
update(11);
```

With a wildcard, you have to use an alias, and I kind of like it, because it makes the rest of the code clearer:

```
import { create, update } from './student_services';
```

Angular is a framework for building client applications in HTML in TypeScript that compiles to JavaScript.



Every Angular Application have the following features.

1. Modules
2. Component
3. Templates
4. Metadata
5. Directives
6. Services
7. Pipes

In angular we create the class and decorate with decorator.

Without the **class decorator**, AppComponent is just a class. There is nothing Angular about it. It is the decorator, which tells angular how to treat the class.

For Example, @Component decorator tells the Angular that the Class is a Component. Similarly, a @Directive a tells the Angular that the class is a Directive. Angular currently has following class decorators

1. @Component
2. @Directive
3. @Injectable
4. @NgModule
5. @Pipe

every decorator expects some metadata to process the class.

Metadata

Metadata tells Angular how to process a class. It provides information about the class.

Modules

Angular apps are modular and Angular has its own modularity system called *NgModules*.

Every Angular app has at least one NgModule class, the root module, conventionally named `AppModule`.

. While the root module _may be the only module in a small application, most apps have many more _feature modules.

An NgModule, whether a root or feature, is a class with an `@NgModule` decorator.

NgModule is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:

- declarations - the view classes that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.
- exports - the subset of declarations that should be visible and usable in the component templates of other modules.
- imports - other modules whose exported classes are needed by component templates declared in this module.
- providers - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app.
- bootstrap - the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

Components

A component controls a patch of screen called a view.

Component's application logic—what it does to support the view—inside a class. The class interacts with the view through an API of properties and methods.

A template is a form of HTML that tells Angular how to display the component.

A template looks like regular HTML, except for a few differences.


```

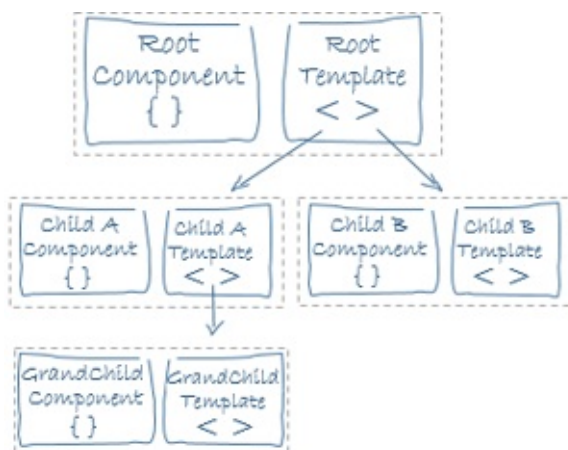
<table>

  <tr><td>Id</td><td>Name</td></tr>
  <tr *ngFor="let student of students">
    <td>{{student.id}}</td>
    <td>{{student.name}}</td>
  </tr>

</table>

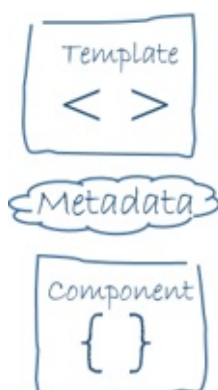
```

a component can use the other component.



Here are a few of the most useful `@Component` configuration options:

- `selector` : CSS selector that tells Angular to create and insert an instance of this component where it finds a `<student-list>` tag in *parent* HTML. For example, if an app's HTML contains `<student-list></student-list>`, then Angular inserts an instance of the `StudentListComponent` view between those tags.
- `templateUrl` : module-relative address of this component's HTML template,



Directives

Angular templates are *dynamic*. When Angular renders them, it transforms the DOM according to the instructions given by **directives**.



A directive is a class with a `@ Directive` decorator. A component is a *directive-with-a-template*; a `@ Component` decorator is actually a `@ Directive` decorator extended with template-oriented features.

Services

Service is a broad category encompassing any value, function, or feature that your application needs.

A service is typically a class with a narrow, well-defined purpose.



It should do something specific and do it well. Examples include:

- logging service
- data service
- message bus
- tax calculator
- application configuration.

Component classes should be lean. They don't fetch data from the server, validate user input, or log directly to the console. They delegate such tasks to services.

A component's job is to enable the user experience and nothing more. It mediates between the view (rendered by the template) and the application logic (which often includes some notion of a *model*). A good component presents properties and methods for data binding. It delegates everything nontrivial to services.

Pipes

Pipes are used to transform the data from one format to another format.

A component has a lifecycle managed by Angular.

Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Angular offers **lifecycle hooks** that provide visibility into these key life moments and the ability to act when they occur.

A directive has the same set of lifecycle hooks, minus the hooks that are specific to component content and views.

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them.

| | |
|----------------------------|---|
| <code>ngOnChanges()</code> | Respond when Angular (re)sets data-bound input properties. The method receives a <code>SimpleChanges</code> object of current and previous property values. Called before <code>ngOnInit()</code> and whenever one or more data-bound input properties change. |
| <code>ngOnInit()</code> | Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called <i>once</i> , after the <i>first</i> <code>ngOnChanges()</code> . |
| <code>ngOnDestroy</code> | Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called <i>just before</i> Angular destroys the directive/component. |

Components are a feature of Angular that let us create a new HTML language and they are how we structure Angular applications.

HTML comes with a bunch of pre-built tags like `<input>` and `<form>` which look and behave a certain way. In Angular we create new custom tags with their own look and behaviour.

An Angular application is therefore just a set of custom tags that interact with each other, we call these tags Components.

Component is a combination of a view (the template) and some logic (our TS class).

Let's create a class:

```
export class StudentListComponent {  
}
```

Our application itself is a simple component. To tell Angular that it is a component, we use the `@Component` decorator. To be able to use it, we have to import it:

```
import { Component } from '@angular/core';  
@Component()  
export class StudentListComponent {  
}
```

If you're new to TypeScript then the syntax of this next statement might seem a little foreign:

```
@Component({  
  // ...  
})
```

These are called decorators. decorators as metadata added to our code.

When we use `@Component` on the `HelloWorld` class, we are “decorating” `StudentListComponent` as a Component.

We want to be able to use this component in our markup by using a `<student-list>` tag.

To do that, we configure the `@Component` and specify the selector `student-list`.

```
1 @Component({  
2   selector: 'student-list'  
3   // ... more here  
4 })
```

The selector property here indicates which DOM element this component is going to use.

In this case, any `<student-list></student-list>` tags that appear within a template will be compiled using the `StudentListComponent` class and get any attached functionality.

Adding a template with templateUrl

In our component we are specifying a `templateUrl` of `./student-list.component.html`.

This means that we will load our template from the file `student-list.component.html` in the same directory as our component.

Adding a template

We can define templates two ways, either by using the `template` key in our `@Component` object or by specifying a `templateUrl`.

We could add a template to our `@Component` by passing the `template` option:

```
@Component({
  selector: 'student-list',
  template: `
    <p>
      //logic
    </p>
  `
})
```

In a web application, we need to display data on an HTML page and read the data from input controls on an HTML page.

In Angular, everything is a component; the HTML page is represented as a template, and it is always associated with a Component class.

Application data lives on the component's class properties.

Displaying data

we have multiple syntaxes to display the data in the angular.

Interpolation syntax

The double curly braces are the interpolation syntax in Angular. we also call it as interpolation.

```
Ex: -  
{{message}}
```

For any property on the class that we need to display on the template, we can use the property name surrounded by double curly braces. Angular will automatically render the value of the property in the browser.

we can bind the message property to a text box.

```
template: `  
    <h1>{{message}}</h1>  
    <input type="text" value="{{message}}"/>
```

Notice that the preceding template is a multiline string, and it is surrounded by (backtick) symbols instead of single or double quotes.

Interpolation syntax is one-way data binding, and data flows from the data source (Component class) to view (template).

Only the value of the property is updated on the template, it will not happen vice-versa, that is, changes made to controls on the template will not update the property value.

Property binding

Property binding is another form of data binding syntax in Angular.

<element-name [element-property-name] = "component-property-name">

Property binding syntax:

element-name: This can be any HTML tag, or custom tag

element-property-name: Specifies the property of the corresponding DOM element for the HTML tag or custom tag property name surrounded by square brackets

component-property-name: Specifies the property of the component class or expression.

```
template: `
    <h1 [textContent]="message"></h1>
    <input type="text" [value]="message"/>`
```

Instead of using interpolation syntax, we are wrapping the `textContent` property of the `<h1>` tag and `value` property input tag in square braces, and on the right side of this expression, we are assigning the Component class properties. The output will be the same as when we are using interpolation syntax.

Property binding syntax is also one-way data binding, data flows from data source (Component class) to view (template).

Attribute binding

Angular always uses properties to bind the data. But if there is no corresponding property for the attribute of an element, Angular will bind data to attributes. Attribute binding syntax starts with the keyword `attr` followed by the name of the attribute and then assigns it to the property of the Component class or an expression:

```
<td [attr.colspan]="colSpanValue"></td>
```


Using event binding syntax, we can bind built-in HTML element events, such as click, change, blur, and so on, to Component class methods. We can also bind custom events on components or directives,

Event binding syntax uses parenthesis symbols (). We need to surround the event property name with parenthesis symbols () on the left side of the expression, on the right side we will specify one of the Component methods which will be invoked when the event is triggered.

```
export class AppComponent {  
  
    public message: string = 'Angular - Event Binding';  
  
    showMessage() {  
        alert("You pressed a key on keyboard!");  
    }  
  
}
```

We have added a method named showMessage() to the AppComponent class, this method will be invoked whenever we type a key in the text box.

The code for the revised template on AppComponent is as follows:

```
template: `  
    <h1>{{message}}</h1>  
    <input type="text" [value]="message" (keypress)="showMessage()"/>`
```

We have added a keypress event surrounded by parenthesis symbols on the text box to bind with the showMessage() method in the AppComponent class.

The code for src/app.component.ts is as follows:

```
@Component({
  selector: 'event-binding-app',
  template: `
    <p>{{message}}</p>
    <input type="text" (keypress)="showMessage($event)"/>
  `
})
export class AppComponent {

  public message: string = 'Angular - Event Binding';

  showMessage(onKeyPressEvent) {
    this.message = onKeyPressEvent.target.value;
  }

}
```

- To the showMessage method, we are passing a special Angular \$event object \$event keyword represents the current DOM event object
- On the AppComponent class showMessage method, we are accepting \$event passed from template into the onKeyPressEvent method parameter
- Every DOM event object has a target property, which represents the DOM element on which the current event is raised
- We are using the onKeyPressEvent.target object, which represents the text box
- We are using the onKeyPressEvent.target.value property to access to the text box value
- We are assigning the value of the text box to the message property

Directives

The Angular directive helps us to manipulate the DOM. You can change the appearance, behaviour or a layout of a DOM element using the Directives. They help you to extend HTML.

Directives are used to extend HTML, teaching new tricks to HTML elements without disturbing their actual structure and implementation. They are quite useful in scenarios where the application wants certain elements to behave in a given way, depending on a value in the model.

There are three kinds of directives in Angular:

1. Component Directive
2. Structural directives
3. Attribute directives

1 . Component Directive

Components are special directives in Angular. these are the basic build blocks in angular. it is always associated with the template.

2 . Structural directives

Structural directives can change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by Asterix symbol.

3 . Attribute directives

An Attribute or style directive can change the appearance or behaviour of an element.

Angular provides a number of built-in directives, which are attributes we add to our HTML elements that give us dynamic behavior. Angular comes with very few directives, the remaining directives in AngularJS 1 are replaced with new concepts of Angular.

Structural Directives

The structural directives allow us to change the DOM structure in a view by adding or removing elements. In this section, we will explore built-in structural directives, `ngIf`, `ngFor`, and `ngSwitch`.

`ngIf`

The `ngIf` directive is used when you want to display or hide an element based on a condition. The condition is determined by the result of the expression that you pass into the directive.

The `ngIf` directive is used for adding or removing elements from DOM dynamically:

```
<element *ngIf="condition"> content </element>
```

If the condition is true, Angular will add content to DOM, if the condition is false it will physically remove that content from DOM:

```
<div *ngIf="isReady">
  <h1>Structural Directives</h1>
  <p>They lets us modify DOM structure</p>
</div>
```

when `isReady` value is true, the content inside the `<div>` tag will be rendered on the page, whenever it is false, both tags inside the `<div>` tag will be completely removed from DOM. The asterisk (*) symbol before `ngIf` is a must.

Scenarios: -

```
<div *ngIf="false"></div> <!-- never displayed -->
<div *ngIf="a > b"></div> <!-- displayed if a is more than b -->
<div *ngIf="str == 'yes'"></div> <!-- displayed if str is the string "yes" -->
<div *ngIf="myFunc()"></div> <!-- displayed if myFunc returns truthy -->
```

`ngSwitch`

Sometimes you need to render different elements depending on a given condition. When you run into this situation, you could use `ngIf` several times like this:

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar != 'A' && myVar != 'B'">Var is something else</div>
</div>
```

But as you can see, the scenario where `myVar` is neither A nor B is verbose when all we're trying to express is an else.

For cases like this, Angular introduces the `ngSwitch` directive.

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchCase="'A'">Var is A</div>
  <div *ngSwitchCase="'B'">Var is B</div>
  <div *ngSwitchDefault>Var is something else</div>
</div>
```

Ex:-

```
<div class="ui raised segment">
  <ul [ngSwitch]="choice">
    <li *ngSwitchCase="1">First choice</li>
    <li *ngSwitchCase="2">Second choice</li>
    <li *ngSwitchCase="3">Third choice</li>
    <li *ngSwitchCase="4">Fourth choice</li>
    <li *ngSwitchCase="2">Second choice, again</li>
    <li *ngSwitchDefault>Default choice</li>
  </ul>
</div>
```

ngFor

The `ngFor` is a repeater directive, it's used for displaying a list of items. We use `ngFor` mostly with arrays in JavaScript, but it will work with any iterable object in JavaScript. The `ngFor` directive is similar to the `for...in` statement in JavaScript.

The role of this directive is to repeat a given DOM element (or a collection of DOM elements) and pass an element of the array on each iteration.

example:

```
public frameworks: string[] = ['Angular', 'React', 'Ember'];
```

The framework is an array of frontend framework names. Here is how we can display all of them using ngFor:

```
<ul>
  <li *ngFor="let framework of frameworks">
    {{framework}}
  </li>
</ul>
```

The preceding code uses ngFor to display the list of framework names. Let us understand each part of the ngFor syntax:

```
*ngFor="let framework of frameworks"
```

There are multiple segments in the ngFor syntax, which are *ngFor, let framework, and frameworks. We will now see them in detail:

- frameworks: This is a array and data source for the ngFor directive on which it will iterate.
- let framework: let is a keyword used for declaring the template input variable. The template input variable represents a single item in the list during iteration. We can use a framework variable inside an ngFor template to refer to the current item of iteration.
- *ngFor: ngFor represents the directive itself, the asterisk (*) symbol before ngFor is a must.

We can also iterate through an array of objects like these:

```
this.people = [
  { name: 'Ram', age: 35, area: 'AmeerPet' },
  { name: 'Robert', age: 12, area: 'S R Nagar' },
  { name: 'Raheem', age: 22, area: 'Yousuf Guda' }
];
```

And then render a table based on each row of data:

```
<h4 class="ui horizontal divider header">
  List of objects
</h4>
<table class="ui celled table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>Area</th>
    </tr>
  </thead>
  <tr *ngFor="let p of people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
    <td>{{ p.area }}</td>
  </tr>
</table>
```

O/P will be: -

| Name | Age | Area |
|--------|-----|-------------|
| Ram | 35 | Ameerpet |
| Robert | 12 | S R Nagar |
| Raheem | 22 | Yousuf Guda |

Getting an index: -

There are times that we need the index of each item when we're iterating an array.

We can get the index by appending the syntax `let idx = index` to the value of our `ngFor` directive, separated by a semi-colon.

```
<tr *ngFor="let p of people;let i=index;">
  <td>{{i+1}}</td>
  <td>{{ p.name }}</td>
  <td>{{ p.age }}</td>
  <td>{{ p.area }}</td>
</tr>
```

o/p will be: -

| No | Name | Age | Area |
|----|--------|-----|-------------|
| 1 | Ram | 35 | Ameerpet |
| 2 | Robert | 12 | S R Nagar |
| 3 | Raheem | 22 | Yousuf Guda |

The attribute directives allow us to change the appearance or behavior of an element.

we have two built-in attribute directives, `ngStyle`, and `ngClass`.

ngStyle

The `ngStyle` directive is used when we need to apply multiple inline styles dynamically to an element.

With the `NgStyle` directive, you can set a given DOM element CSS properties from Angular expressions.

The simplest way to use this directive is by doing `[style.<cssproperty>]="value"`.

For example:

```
<div [style.background-color]='yellow'>
  Uses fixed yellow background
</div>
```

This snippet is using the `NgStyle` directive to set the `background-color` CSS property to the literal

string `'yellow'`.

Another way to set fixed values is by using the `NgStyle` attribute and using key value pairs for each property you want to set.

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
  Uses fixed white text on blue background
</div>
```

But the real power of the `NgStyle` directive comes with using dynamic values.

```
<p [ngStyle]="getInlineStyles(framework)">{{framework}}</p>

// in the component class
getInlineStyles(framework) {
  let styles = {
    'color': framework.length > 3 ? 'red' : 'green',
    'text-decoration': framework.length > 3 ? 'underline' : 'none'
  };
  return styles;
}
```

ngClass

The NgClass directive, represented by a ngClass attribute in your HTML template, allows you to dynamically set and change the CSS classes for a given DOM element.

Ex: -

```
.red {  
  color: red;  
  text-decoration: underline;  
}  
.bolder {  
  font-weight: bold;  
}
```

In the component class,

```
geClasses(framework) {  
  let classes = {  
    red: framework.length > 3,  
    bolder: framework.length > 4  
  };  
  return classes;  
}
```

In the template,

```
<p [ngClass]="geClasses(framework)">{{framework}}</p>
```

Custom Directives

Custom Directives can be used to add behavior such as a particular style property according to a value in the model, handling of events, and any additional behavior on the element that enhances its functionality.

Building a Custom Attribute Directive

1 . Creating the class and provide the annotation `@Directive`.

```
@Directive({
  selector: '[highlight]'
})
export class HighLightDirective {

}
```

2 . Register the directive in the declarations array of the ngmodule metadata.

```
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, HighLightDirective],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

To change the element behaviour we need to get the current element first.

For this purpose we use need use the dependency injection.

```
@Directive({
  selector: '[highlight]'
})
export class HighLightDirective {

  constructor(private element: ElementRef) {
  }
}
```

The directive can listen to the events of its container elements by using the `HostListener`

```
@HostListener('mouseenter')
onMouseEnter() {
    this.setAppearance(this.backgroundColor, 'pointer');
}
```

the directive can also listens the attributes of the element by using the @Input

```
@Input('highlight') backgroundColor: string;
```

```
<li *ngFor="let user of users" highlight="#000000">
```

Complete directive definition

```
@Directive({
    selector: '[highlight]'
})
export class HighLightDirective {

    @Input('highlight') backgroundColor: string;

    constructor(private element: ElementRef) {
    }

    @HostListener('mouseenter') onMouseEnter() {
        this.setAppearance(this.backgroundColor, 'pointer');
    }

    @HostListener('mouseleave') onMouseLeave() {
        this.setAppearance(null, null);
    }

    setAppearance(color: string, cursor: string) {
        let style = this.element.nativeElement.style;
        style.backgroundColor = color;
        style.cursor = cursor;
    }
}
```

The real-world applications will be complex, and they will have multiple components. We are going to rewrite our application to use multiple components and understand how these components communicate with each other.

Input Properties

Inputs specify the parameters we expect our component to receive. To designate an input, we use the `@Input()` decoration on a component class property.

Inputs specify the parameters we expect our component to receive.

It is available in the `@angular/core` package.

Ex1

```
import { Component, Input } from '@angular/core';
import { Student } from '../student';
@Component({
  selector: 'student-details',
  templateUrl: './student-details.component.html',
})
export class StudentDetailsComponent {
  @Input() student: Student;
}
```

Now the student property of the StudentDetailsComponent class is available for property binding.

```
<div *ngFor="let s of students">
  <student-details [student]="s"></student-details>
</div>
```

Aliasing input properties:-

```
@Input('studentData') student: Student;
```

```
<div *ngFor="let s of students">
  <student-details [studentData]="s"></student-details>
</div>
```

Output Properties

Output properties are used by parents listens to child events.

The child component exposes an EventEmitter property with which it emits events when something happens. The parent binds to that event property and reacts to those events.

When Child Component needs to communicate with the parent it raises the event. The Parent Component listens to that event and reacts to it.

EventEmitter Property

To Raise an event, the component must declare an EventEmitter Property. The Event can be emitted by calling the .emit() method.

```
countChanged: EventEmitter<number> = new EventEmitter();
```

And then call emit method passing the whatever the data you want to send as shown below.

```
this.countChanged.emit(this.count);
```

@Output Decorator

Using the EventEmitter Property gives the components ability to raise an event. But to make that event accessible from parent component, you must decorate the property with @Output decorator.

In the child component

1 . Declare a property of type EventEmitter and instantiate it

```
countChanged: EventEmitter<number> = new EventEmitter();
```

2 . Mark it with a @Output annotation

```
@Output() countChanged: EventEmitter<number> = new EventEmitter();
```

3 . Raise the event passing it with the desired data

```
this.countChanged.emit(this.count);
```

In the Parent Component

1 . Bind to the Child Component using event binding and listen to the child events

```
<child-component [count]=ClickCounter (countChanged)="countChangedHandler($event)">
</child-component>
```

2 . Define the event handler function

```
countChangedHandler(count: number) {
  this.ClickCounter = count;
  console.log(count);
}
```

Pipes

Pipes takes data as input and transforms it to the desired output

Pipes are used to transform data, when we only need that data transformed in a template.

If we need the data transformed generally we would implement it in our model, for example we have a number 1234.56 and want to display it as a currency such as \$1,234.56.

We could convert the number into a string and store that string in the model but if the only place we want to show that number is in a view we can use a pipe instead.

We use a pipe with the | syntax in the template, the | character is called the pipe character.

```
{{ 1234.56 | currency }}
```

O/P:

USD1,234.56.

A pipe can accept optional parameters to modify the output. To pass parameters to a pipe, simply add a colon and the parameter value to the end of the pipe expression:

```
pipeName: parameterValue
```

Ex:

```
{{ 1234.56 | currency : 'USD' }}
```

O/P:

USD1,234.56.

You can also pass multiple parameters this way:

```
pipeName: parameter1: parameter2
```


Pipes provided by Angular

Angular provides the following set of built-in pipes.

CurrencyPipe

This pipe is used for formatting currencies.

Its first argument is an abbreviation of the currency type (e.g. "EUR", "USD", and so on).

```
{{ 1234.56 | currency:'GBP' }}
```

The above prints out GBP1,234.56.

instead of the abbreviation of GBP we want the currency symbol to be printed out we pass as a second parameter the boolean true.

```
{{ 1234.56 | currency:"GBP":true }}
```

The above prints out £1,234.56.

DatePipe

This pipe is used for the transformation of dates.

The first argument is a format string.

```
<div class="card card-block">
  <h4 class="card-title">Date</h4>
  <div class="card-text">
    <p >{{ dateVal | date: 'shortTime' }}</p> ①
    <p>{{ dateVal | date: 'shortTime' }}</p>
    <p >{{ dateVal | date: 'fullDate' }}</p>
    <p>{{ dateVal | date: 'fullDate' }}</p>
    <p ngNoBindable>{{ dateVal | date: 'd/M/y' }}</p>
    <p>{{ dateVal | date: 'd/M/y' }}</p>
  </div>
</div>
```

DecimalPipe

This pipe is used for transformation of decimal numbers.

The first argument is a format string of the form

```
{minIntegerDigits}. {minFractionDigits}-{maxFractionDigits}
```

Ex:

```
<div class="card card-block">
  <div class="card-text">
    <h4 class="card-title">DecimalPipe</h4>
    <p>{{ 3.14159265 | number: '3.1-2' }}</p>
    <p>{{ 3.14159265 | number: '1.4-4' }}</p>
  </div>
</div>
```

O/P:

003.14

3.1415

JsonPipe

This transforms a JavaScript object into a JSON string.

```
<div class="card card-block">
  <h4 class="card-title">JsonPipe</h4>
  <div class="card-text">
    <p>{{ student }}</p>
    <p>{{ student | json }}</p>
  </div>
</div>
```

O/P

[Object Object]

```
{ "id":1,"name":"RK"}
```

LowerCasePipe and UpperCasePipe

transforms the input into the lowercase and uppercase.

PercentPipe

Formats a number as a percent.

```
<div class="card card-block">
  <h4 class="card-title">PercentPipe</h4>
  <div class="card-text">
    <p>{{ 0.123456 | percent }}</p>
    <p>{{ 0.123456 | percent: '2.1-2' }}</p>
    <p>{{ 0.123456 | percent : "3.4-4" }}</p>
  </div>
</div>
```

O/P

12.346%

12.35%

012.3456%

SlicePipe

This returns a slice of an array. The first argument is the start index of the slice and the second argument is the end index.

If either indexes are not provided it assumes the start or the end of the array and we can use negative indexes to indicate an offset from the end.

```
<div class="card card-block">
  <h4 class="card-title">SlicePipe</h4>
  <div class="card-text">
    <p>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>
    <p>{{ [1,2,3,4,5,6] | slice:2 }}</p>
    <p>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>
  </div>
</div>
```

O/P

2,3

3,4,5,6

3,4,5

Angular allows you to create your own custom pipes based on your project requirement.

Each custom pipe implementation must

- have the `@Pipe` decorator with pipe metadata that has a name property. This value will be used to call this pipe in template expressions. It must be a valid JavaScript identifier.
- implement the `PipeTransform` interface's `transform` method. This method takes the value being piped and a variable number of arguments of any type and return a transformed ("piped") value.

Pipe decorator

To create a pipe we use the `@Pipe` decorator and annotate a class like so:

```
import { Pipe } from '@angular/core';  
.  
.  
.  
@Pipe({  
  name: "default"  
})  
class DefaultPipe { }
```

The name parameter for the Pipe decorator is how the pipe will be called in templates.

Transform function

The actual logic for the pipe is put in a function called `transform` on the class.

```
class DefaultPipe implements PipeTransform{  
  transform(value: string, fallback: string): string {  
    let image = "";  
    if (value) {  
      image = value;  
    } else {  
      image = fallback;  
    }  
    return image;  
  }  
}
```

Usage in the template

```
<img [src]="imageUrl | default:'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg'"/>
```

- value gets passed `imageUrl` which is blank.

- fallback gets passed '<http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg>'

Pipes are a way of having a different visual representation for the same piece of data without storing unnecessary intermediate data on the component.

Services

Service is a piece of reusable code with a Focused Purpose. A code that you will use it in many components across your application. service is a TypeScript class. It can be injected into any component, directive, or service and can be used whenever needed.

What services are used for?

1. Features that are independent of components such a logging services
2. Share logic across components
3. Encapsulate external interactions like data access

Advantageous of Service

1. Services are easier Test.
2. Services are easier to Debug.
3. You can reuse the service.

creating the service in angular.

1 . create a class

```
export class StudentService {  
    // logic  
}
```

2 . provide the @Injectable decorator.

```
import { Injectable } from '@angular/core';  
  
@Injectable()  
export class StudentService {  
    // logic  
}
```

3 . register the service in the providers array in the module metadata.

```
import { StudentService } from './student/student.service';
@NgModule({
  imports: [...],
  declarations: [...],
  providers:[StudentService],
  bootstrap: [...],
})
export class AppModule { }
```

1. using the service in the component

```
export class StudentListComponent implements OnInit {

  students:Student[];

  constructor(private studentService : StudentService) {
    this.studentService.getStudents().subscribe(
      (data) => this.students = data
    )
  }

}
```

In the above code we are using the StudentService in the student list component.

here we are using the dependency injection for the service class objection.

Dependency Injection

Dependency Injection (DI) is a technique in which we provide an instance of an object to another object, which depends on it. This technique is also known as “**Inversion of Control**” (IoC)

software component should not create its dependencies; rather, the dependencies should be injected by an external source. The external source should know how objects are created, which reduces the chance of duplicating code for creating objects. The external source is called Inversion of Control container or, in short, the IoC container.

A registered dependency can be injected into a component's constructor.

Angular's DI system is based on injectors. Angular creates an injector when the application is bootstrapped. All the application dependencies are registered in the injector, and the injector serves them when they are requested. The injector can be configured by registering providers at the module level or at a component level.

1 . we can inject the service into component.

```
export class StudentListComponent implements OnInit {  
  
    students:Student[];  
  
    constructor(private studentService:StudentService) {  
  
    }  
  
}
```

2 . we can inject multiple services into the single component.

```
export class StudentListComponent implements OnInit {  
  
    students:Student[];  
  
    constructor(private studentService:StudentService,private addressservice:AddressService) {  
  
    }  
  
}
```


3 . we can inject one service into another service.

```
import { Http } from "@angular/http";
@Injectable()
export class StudentService {

    constructor(private http:Http) { }

    getStudents()
    {
        return this.http.get("url");
    }
}
```

Http

Angular comes with its own HTTP library which we can use to call out to external APIs.

When we make calls to an external server, we want our user to continue to be able to interact with the page. That is, we don't want our page to freeze until the HTTP request returns from the external server. To achieve this effect, our HTTP requests are asynchronous.

HTTP has been split into a separate module in Angular. This means that to use it you need to import from `@angular/http`.

```
import {Http} from "@angular/http";
```

```
import {HttpModule} from "@angular/http";
```

Http class expose a connection method named `request()` to perform any type of http connection.

Angular team has created some syntax shortcuts for the most common request operations, such as GET, POST, PUT, and every existing HTTP.

```
var requestOptions = new RequestOptions({
    method: RequestMethod.Get,
    url: '/spring/users'
});
var request = new Request(requestOptions);
var myHttpRequest: Observable<Response> = http.request(request);
```

Instead we can use

```
let myHttpRequest: Observable<Response> = http.get('/spring/users');
```

Http class connection methods operate by returning an Observable stream of Response object instances. That is why we need to write the following code to process the response.

```
myHttpRequest.map(response: Response => response.json())
    .subscribe(data => console.log(data));
```

When to use the Request and RequestOptionsArgs classes

we should use these classes, when we want to send special HTTP headers in your requests or append query string parameters automatically to each request.

1 . Think of a use case where you want to add an authentication token to each request in order to prevent unauthorized users from reading data from one of your API endpoints.

2 . for pagination we need to send the same filter criteria and the page information for each and every request in the list page.

In the following example, we read an authentication token and append it as a header to our request to a data service. Contrary to our example, we will inject the options hash object straight into the Request constructor, skipping the step of creating a RequestOptions object instance. Angular 2 provides a wrapper class for defining custom headers as well, and we will

take advantage of it in this scenario. Let's figure out that we do have an API that expects all requests to include a custom header named Authorization, attaching the authToken that we received when logging into the system, which was then persisted in the browser's local storage layer, for instance:

```
var authToken = window.localStorage.getItem('auth_token');
var headers = new Headers();
headers.append('Authorization', `Token ${authToken}`);
var request = new Request({
    method: RequestMethod.Get,
    url: '/spring/users',
    headers: headers
});
var authRequest: Observable<Response> = http.request(request);
```

Response object

HTTP requests performed by the Http class return an observable stream of Response class instances.

In our first example, we mapped the content of the stream as a stream of JSON objects by executing the json() method. This method parses the response body as a JSON object, or raises an exception if such a body cannot be parsed. Besides this method, the Response class exposes the text() method, which will parse and return the response body as a plain string.

```
http.get('/spring/users')  
  .map(res: Response => res.txt)  
  .subscribe(data => this.students= data);
```

Handling errors when performing Http requests

Handling errors raised in our requests by inspecting the information returned in the Response object.

We can

double-check that by inspecting the status property to understand the error code or the type property, which can assume the following values: basic, cors, default, error, or opaque.

Inspecting the response headers and the statusText property of the Response object will provide insightful information about the origin of the error.

```
http.get('/spring/users')  
  .map(res: Response => res.txt)  
  .subscribe(data => this.students= data);  
  .catch(error: Response => console.error(error));
```

Http Class simplifies the request and response cycle at the client by providing the methods for each request type. it have the following methods to get the data from the server.

- 1 . get
- 2 . post
- 3 . put
- 4 . delete

Making a Get request

get request used to access the data from the server.

```
http.get('/spring/users')  
  .map(res: Response => res.txt)  
  .subscribe(data => this.students= data);  
  .catch(error: Response => console.error(error));
```

Making a Post request

post request used to send the data to the server.

```
http.post('/spring/users',user)
.map(res: Response => res.txt)
.subscribe(data => this.students= data);
.catch(error: Response => console.error(error));
```

Making a Put request

put request used to update the data in the server.

```
http.put('/spring/users',user)
.map(res: Response => res.txt)
.subscribe(data => this.students= data);
.catch(error: Response => console.error(error));
```

Making a delete request

delete request used to delete the data in the server.

```
http.delete('/spring/users/'+user.id)
.map(res: Response => res.txt)
.subscribe(data => this.students= data);
.catch(error: Response => console.error(error));
```

Service with http request calls.

1 . import the Http class from the @angular/http package.

```
import { Http } from "@angular/http";
```

2 . declare the dependency in the service constructor.

```
constructor(private http:Http) { }
```

3 . provide the required methods and make the http calls

```
insertStudent(student:Student){
    return this.http.post("http://localhost:3000/students",student).map(
        (response) => response.json()
    );
}
```

4 . complete service class will be look like this.

```
import { Injectable } from '@angular/core';

import { Http } from "@angular/http";

import { Student } from "../student";

import { Observable } from "rxjs";

import "rxjs/add/operator/map";

@Injectable()
export class StudentService {

  constructor(private http:Http) { }

  insertStudent(student:Student){
    return this.http.post("http://localhost:3000/students",student).map(
      (response) => response.json()
    );
  }
  getStudents()
  {
    return this.http.get("http://localhost:3000/students").map(
      (response) => response.json()
    );
  }
  getStudentById(id:number)
  {
    return this.http.get("http://localhost:3000/students/"+id).map(
      (response) => response.json()
    );
  }
  deleteStudent(student:Student){
    return this.http.delete("http://localhost:3000/students/"+student.id).map(
      (response) => response.json()
    );
  }
  updateStudent(student:Student){
    return this.http.put("http://localhost:3000/students/"+student.id,student).map(
      (response) => response.json()
    );
  }
}
```

5 . complete component class

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

import { StudentService } from "../student.service";

import { Student } from "../student";

@Component({
  moduleId: module.id,
  selector: 'student-list',
  templateUrl: 'student-list.component.html'
})

export class StudentListComponent{

  students:Student[];

  constructor(private studentService:StudentService) {
    this.students = [];
  }

  load(){
    this.studentService.getStudents().subscribe(
      (data)=>this.students=data,
      (error)=>console.log(error)
    );
  }

}
```

6 . template for the component

```
<button (click)="load();">Load</button>
<a [routerLink]="['/insert']">Insert</a>
<table>

  <tr><td>id</td><td>name</td><td>email</td></tr>

  <tr *ngFor="let student of students">
    <td>{{student.id}}</td><td>{{student.name}}</td><td>{{student.email}}</td>
    <td><a [routerLink]="['/edit',student.id]">Edit</a></td>
    <td><a [routerLink]="['/delete',student.id]">Delete</a></td>
  </tr>

</table>
```


Forms are the key to any web application. Here are a couple of things we do with forms.

- Capturing input from the user
- Validating the user input
- Responding to events
- Displaying the information messages
- Displaying the error messages

In Angular 2 applications (and, hence, in Web Applications), the Form plays a crucial role while developing a UI, as forms provide a way for the users to interact with the application.

Angular has the following two approaches for dealing with forms:

- Template driven forms
- Reactive forms

Before we learn in detail with the forms approach, we need to learn few details about the following angular form classes.

FormControl

Control is the smallest unit in any form; it represents a single form input element (textbox, dropdown, radio button, checkbox, and so on). Control is the fundamental building block of forms API in Angular; a control object encapsulates the input field's value and its state. It is represented using the FormControl class.

Creating a form control

The following code snippet creates a single control named firstName:

```
let firstName = new FormControl();
```

The following code snippet creates a single control named firstName and initializes it with an empty default value:

```
let firstName = new FormControl('');
```

The following code snippet creates a single control named firstName, and initializes with default value.

```
let firstName = new FormControl('RK');
```

Accessing the value of an input control

```
let firstNameValue = firstName.value;
```

Setting the value of input control

We cannot use value property to set the value of form control; it is just a getter. We should use the `setValue()` method to set the value programmatically:

```
firstName.setValue('Ram');
```

Resetting the value of an input control

The `reset()` method on the form control sets the value to null:

```
firstName.reset();
```

Input control states:-

Every input control on an Angular form and the form itself maintains different states depending on the user input and interaction with it:

```
//form control error list object
let errors = firstName.errors
// form control value is valid, it has no errors
let isValid = firstName.valid
// form control value is invalid, it has errors
let isInvalid = firstName.invalid
//Control has been visited
let isTouched = firstName.touched
//Control has not been visited
let isUntouched = firstName.untouched
//Form control's value has changed
let valueChanged = firstName.dirty
//Form control's value has not changed
let valueNotChanged = firstName.pristine
```

FormGroup

Even a simple form contains more than one control that might be dependent on each other. Instead of working with each control and iterating over them to know the value and state of each control and form, we want to know the state of multiple controls at once. Sometimes it makes more sense to think of a series of form controls as a group.

For example, we need a user address which contains the street, city, state, country, and zip code. We can create five individual `FormControl` objects and work them one at a time, but all together they represent an address where we can use the `FormGroup` class:

```
//create a form group
let address = new FormGroup({
  street: new FormControl(''),
  city: new FormControl(''),
  state: new FormControl(''),
  country: new FormControl(''),
  zip: new FormControl('')
});
```

```
//return an object literal of form group value
let formModel = address.value; //{street: "", city: "", state: "",country: "", zip: ""
}
```

```
//check overall state form state
let errors = address.errors; //null
let isValid = address.valid; //true
let isInValid = address.invalid; //false
let isTouched = address.touched; //false
let isUntouched = address.untouched; //true
let valueChanged = address.dirty; //false
let valueNotChanged = address.pristine; //true
```

```
//set the value of the form group
address.setValue({
  street: '1-3 Strand',
  city: 'London',
  state: '',
  country: 'UK',
  zip: 'WC2N 5BW'
});
```

1 . Model-Driven Forms

- a. The form is designed by importing the ReactiveFormsModule.
- b. The validation logic is separated from the form and handled at the component class level in the code.
- c. Since the validation logic is isolated from the form, it can be unit tested easily.

2 . Template-Driven Form

- a. The form is designed by importing the FormsModule.
- b. The validations are handed in the template .
- c. The unit testing of the validation is tricky.

Validation

Validation of controls in the HTML form can be performed using the following attributes:

1. `required`: Used for specifying the value of input element as mandatory
2. `pattern`: sets the regular expression for the data to be entered in an input element
3. `minlength`: The minimum length of the text entered in the input element
4. `maxlength`: The maximum length of the text entered in the input element

Based on the input validity, validation directives change the state of the `ngModel` and `ngForm`, we can access them using their template reference variables.

every form control object have the following states.

1. `untouched`
2. `touched`
3. `pristine`
4. `dirty`
5. `valid`
6. `invalid`

we use these states to validation messages to the user.

to identify the precise validation failure angular uses `errors` object.

```
<span *ngIf="formcontrolref.invalid&&formcontrolref.errors.required">
  please provide the value
</span>
```

Reactive Forms

Reactive forms (also known as model-driven forms) are the new approach introduced in Angular. In contrast to template driven forms in reactive forms, we will write all the form logic like creating controls, forms, and defining validation rules inside our component classes using the forms API instead of HTML.

In a reactive forms approach, we will directly use the classes `FormControl`, `FormGroup` and `FormBuilder` to create input controls, forms, and apply validation rules inside the Component class.

Creating the reactive form in the component class.

1 . We need to import **ReactiveFormsModule** instead of **FormsModule** to make use of Model Driven Forms. We should also add the **ReactiveFormsModule** to the *imports metadata property array*.

```
import { ReactiveFormsModule } from "@angular/forms";
```

```
@NgModule({
  imports: [BrowserModule,HttpModule,ReactiveFormsModule],
  declarations: [AppComponent,StudentRegComponent],
  providers: [StudentService],
  bootstrap:[AppComponent]
})
export class AppModule { }
```

2 . we need to create the formgroup,formcontrol objects in the component class. these are available in the forms package. so we need to import them first.

```
import { FormGroup,FormControl,Validators } from "@angular/forms";
```

3 . create the formgroup reference variable in the component.

```
studentForm:FormGroup;
```

4 . create the formgroup object in the constructor by passing the necessary information.

```
this.studentForm = new FormGroup({
  id:new FormControl('',[Validators.minLength(4)]),
  name:new FormControl('',[Validators.required])
});
```

.FormGroup constructor expecting the json object.

Here key is the form control name and value is the form control object.

FormControl constructor expecting the default value as first argument and list of validators as second argument.

1. Now our model is ready. Now we need to associate our model to the Template. We need to tell angular that we have a model for the form.

```
<form [formGroup]="studentForm">
```

6 . Next, we need to bind Form fields to the **FormControl** models.

```
Id <input type="text" name="id" formControlName="id"/> <br/>
Name <input type="text" name="name" formControlName="name"/><br/>
```

1. complete form is

```
<form [formGroup]="studentForm">

  Id <input type="text" name="id" formControlName="id"/> <br/>

  Name <input type="text" name="name" formControlName="name"/><br/>

  <button (click)="register();" [disabled]="studentForm.invalid">Register</button>

</form>
```

Reactive Forms Validation

A Validator is a function that checks the **FormControl** or a **FormGroup** and returns a list of errors. If the Validator returns a null means that validation has passed.

Built-in Validators

The Angular provides the following Built-in validators

- **required:** There must be a value

- **minlength**: The number of characters must be more than the value of the attribute.
- **maxlength**: The number of characters must not exceed the value of the attribute.
- **pattern**: The value must match the pattern.

1 . adding the validator to the form control.

```
name: new FormControl('', [Validators.required])
```

2 . adding the validation message.

```
<div *ngIf="!studentForm.controls.name.errors.required">
    Name is required
</div>
```

`studentForm.controls.name__` will return the **FormControl** associated with name form element.

We will display the error message when the `name_.valid _` is false.

Template driven forms

the names suggest, we will write all the logic, like creating form controls, forms, and defining validations inside the template in a declarative manner.

When we are working with template driven forms, we will never directly create FormControl, FormGroup object on our own. Instead we will use ngModel, ngModelGroup, ngForm directives on input controls everything internally handled by Angular.

Using the ngModel directive

To work with individual input controls, we should use the ngModel directive.

```
<input type="text" [(ngModel)]="user.username" name="username">
```

Accessing an input control value using ngModel

The ngModel directive on the input controls represents the model object. To access it we need to export to a template reference variable:

```
<input type="text" [(ngModel)]="user.username" name="username" #usernamectrl="ngModel">
```

above text field bind with the username property of the user object.

Using the ngForm directive

The ngForm directive on the form tag represents the model (FormGroup) object, to access it we need to export to a template reference variable:

```
<form novalidate #formRef="ngForm"></form>
```

Submitting a form using the ngSubmit method

Let us invoke the onSubmit() method whenever an ngSubmit event is triggered:

```
<form novalidate #formRef="ngForm" (ngSubmit)="onSubmit(formRef.value)">
</form>
```

Adding validations: -

use the validation directives in the template driven forms.

```
<input type="text" [(ngModel)]="user.username" name="username" #usernamectrl="ngModel"
      required minlength="6"
>
```

in the above configuration we are using the required validation and minlength validations.

to configure the error messages.

```
<div *ngIf="usernamectrl.touched && usernamectrl?.errors?.required" class="error-messa
ge">
  user name is required.
</div>
```

```
<div class="error-message" *ngIf="firstNameRef.touched && firstNameRef?.errors?.minlen
gth">
  You should enter minimum {{firstNameRef?.errors?.minlength.requiredLength}} charac
ters
  into first name, but you entered only {{firstNameRef?.errors?.minlength.actualLeng
th}}
  characters.
</div>
```

disabling the form submit button when the form is invalid: -

```
<input type="submit" [disabled]="formRef.invalid">Submit</button>
```

configuring the template driven forms

1 . Add the import statement to import the forms module from the angular forms package.

```
import { FormsModule } from "@angular/forms";
```

2 . We should also add the FormsModule to the *imports metadata property array*.

```
imports: [BrowserModule,...,FormsModule,...],
```

.3 . basic form template

```
<form>

  id<input type="text" name="id" [(ngModel)]="student.id"/><br/>
  firstname<input type="text" name="firstname" [(ngModel)]="student.firstname"/><br/>
>

</form>
```

4 . all your form data directly available in the student model object.

Routing

Routing allows you to move from one part of the application to another part or one View to another View.

Navigate to a specific view by typing a URL in the address bar

- Pass optional parameters to the View
- Bind the clickable elements to the View and load the view when user performs application tasks
- Handles back and forward buttons of the browser
- Allows you to dynamically load the view

Components of Router Module

Router

The Angular Router is an object that enables navigation from one component to the next component as users perform application tasks like clicking on menus links, buttons or clicking on back/forward button on the browser. We can access the router object and use its methods like *navigate()* *_or_* *_navigateByUrl()* , to navigate to a route.

Route

Route tells the Angular Router which view to display when a user clicks a link or pastes a URL into the browser address bar. Every route consists of a path and a component it is mapped to. The Router object parses and builds the final URL using the Route.

Routes

Routes is an array of Route objects our application supports.

RouterOutlet

The RouterOutlet is a directive (<router-outlet>) that serves as a placeholder, where the router should display the view.

RouterLink

The RouterLink is a directive that binds the HTML element to a Route. Clicking on the HTML element, which is bound to a RouterLink, will result in navigation to the Route. The RouterLink may contain parameters to be passed to the route's component.

ActivatedRoute

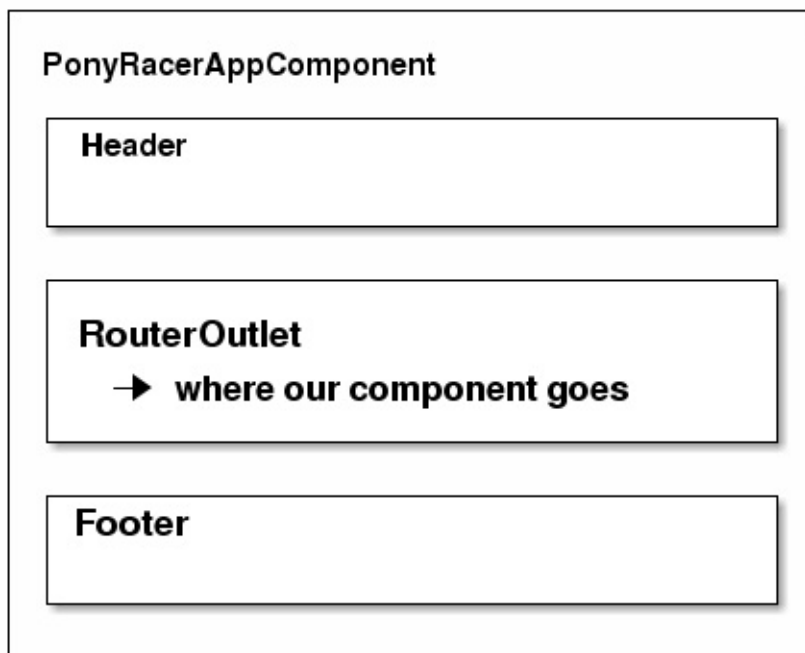
The ActivatedRoute is an object that represents the currently activated route associated with the loaded Component.

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core. It is in its own library package,

`@angular/router` . Import what you need from it as you would from any other Angular package.

```
import { RouterModule, Routes } from '@angular/router';
```

Implementing routing in angular applications.



- 1 . configure the router-outlet tag in the main component(appcomponent).
- 2 . configure the navigation rules. each navigation rule is a combination of the path and component.

```
const routes:Routes=[
  {
    path:'create',
    component:StudentFormComponent
  },
  ....
  ....
]
```

- 3 . configure the routingmodule by using the above configured routes.

```
export const routingModule = RouterModule.forRoot(routes);
```

- 4 . import the routing module in the main module.

```
@NgModule({
  imports: [...,routingModule],
  declarations: [...],
  providers:[...],
  bootstrap: [...],
})
export class AppModule { }
```

5 . configure the anchor tags in the menu by providing the routerLink attribute.

```
<a [routerLink]="['/create']">create</a>
```

Implementing the Route parameters.

We can pass values when navigating from one component to the other.

1. configure the route parameters when you define the route in routes.

```
{
  path: 'delete/:studentId',
  component: StudentDeleteComponent
},
```

- 2 . configure the router link by providing the dynamic value or static value when we are creating the links.

```
<tr *ngFor="let student of students">
  <td><a [routerLink]="['/delete', student.id]">Delete</a></td>
</tr>
```

- 3 . As soon as the component is initialized, we are use ActivatedRoute to access the route parameters using the Params property. inject the ActivatedRoute service to the component class

```
export class StudentDeleteComponent implements OnInit {

  constructor(private route : ActivatedRoute ,private studentService:StudentService)
  {

  }
  .....
  .....
}
```

```
this.route.snapshot.paramMap.get("studentId")
```

- 4 . use the above configuration to get the data from the server.

```
this.student.id=parseInt(this.route.snapshot.paramMap.get("abc"));
this.updateForm = true;
this.studentService.getStudentById(this.student.id).subscribe(
  (data) => this.student = data
)
```


Router

The primary job of the router is to manage navigation between different router states. There are two ways to accomplish this: imperatively, by calling `router.navigate`, or declaratively, by using the `RouterLink` directive.

`Router.navigate` method used to navigate between two states.

```
router.navigate('/edit');
```

Passing matrix params.

go to the specific route: -

```
router.navigate(['/edit', 1]);
```

go to the dynamic route: -

```
router.navigate(['/edit', student.id]);
```

Guards

The router uses guards to make sure that navigation is permitted, which can be useful for security, authorization, and monitoring purposes.

Guards used for the route configuration to handle the following scenarios.

1. User is not authorized to navigate to the target component.
2. User must login (*authenticate*) first.
3. When we should fetch some data before you display the target component.
4. When we want to save pending changes before leaving a component.
5. When we ask the user if it's OK to discard pending changes rather than save them.

A guard's return value controls the router's behavior:

- If it returns `true` , the navigation process continues.
- If it returns `false` , the navigation process stops and the user stays put.

There are four types of guards: `canLoad` , `canActivate` , `canActivateChild` , and `canDeactivate`.

1 . create a class and implements the `CanLoad` interface

```
@Injectable()
class CanLoadContacts implements CanLoad {
    constructor(private permissions: Permissions,
                private currentUser: UserToken) {}
    canLoad(route: Route): boolean {
        if (route.path === "contacts") {
            return this.permissions.canLoadContacts(this.currentUser);
        } else {
            return false;
        }
    }
}
```

2 . register the Guard in the module as a service provider.

```
@NgModule({
    //...
    providers: [CanLoadContacts],
    //...
})
class MailModule {
}
```

3 . use the Guard in the route configuration.

```
{
  path: 'contacts',
  canLoad: [CanLoadContacts],
  component: ContactsComponent
}
```

Using the CanActivate Guards.

```
@Injectable()
class CanActivateContacts implements CanActivate {
  constructor(private permissions: Permissions,
    private currentUser: UserToken) {}
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot){
    if (route.routerConfig.path === "contacts") {
      return this.permissions.canActivate(this.currentUser);
    } else {
      return false;
    }
  }
}
```

Using CanActivateChild

The `canActivateChild` guard is similar to `canActivate`, except that it is called when a child of the route is activated, and not the route itself.

```
@Injectable()
class AllowUrl implements CanActivateChild {
  constructor(private permissions: Permissions,
    private currentUser: UserToken) {}
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot){
    return this.permissions.allowUrl(this.currentUser, state.url);
  }
}
```

Using CanDeactivate

The `canDeactivate` guard is different from the rest. Its main purpose is not to check permissions, but to ask for confirmation.

```
class SaveChangesGuard implements CanDeactivate<ComposeCmp> {  
    constructor(private dialogs: Dialogs) {}  
    canDeactivate(component: ComposeCmp, route: ActivatedRouteSnapshot,  
state: RouterStateSnapshot) {  
        if (component.unsavedChanges) {  
            return this.dialogs.unsavedChangesConfirmationDialog();  
        } else {  
            return Promise.resolve(true);  
        }  
    }  
}
```

