Table of Contents

Angular Framework

Attend the demo at Satya technologies. we have week end and regular batches also.

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.

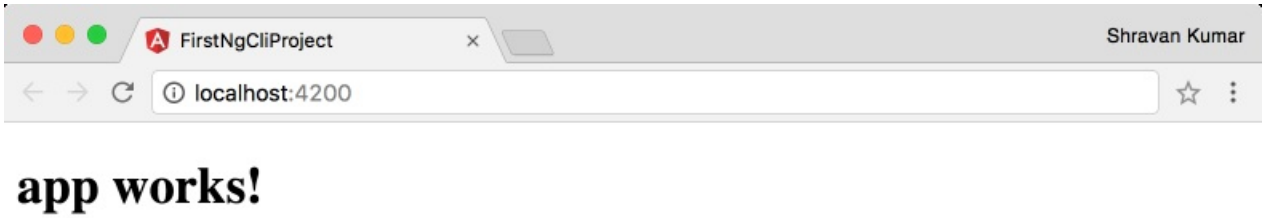
Once you run it, you'll see the following output:

```
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!

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 insta
nce 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 hoiting

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 dont 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 (⇒). 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';
```

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

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 tag will be rendered on the page, whenever it is false, both tags inside the 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>
```

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>
```

When we want to send data from your component to the outside world, we use output bindings.

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.