

Angular 2

One framework for Mobile and desktop.

People matter, results count.

Document History

Date	Course Version No.	Software Version No.	Developer / SME	Reviewer	Approver	Change Record Remarks
24/08/2016	0.1	2.0.0-beta.17	Karthik Muthukrishnan			
13/02/2017	0.2	2.4.1	Karthik Muthukrishnan			Changed the PPT contents to version 2.4.1

Course Goals and Non Goals

➤ Course Goals

- Understand the architecture and fundamentals of Angular 2
- Creating Single page responsive web application using Angular using TypeScript



➤ Course Non Goals

- Comparing Angular 2 with Angular 1
- Creating Angular 2 application using ES5, ES 2015 and Dart

Pre-requisites

- HTML, JavaScript, CSS, Bootstrap basics(optional)

Intended Audience

➤ Web application developers



Day Wise Schedule

➤ Day 1

- Lesson 1 : TypeScript Introduction
- Lesson 2 : Angular 2 Fundamentals

➤ Day 2

- Lesson 3: Databinding
- Lesson 4: Directives and Pipes

➤ Day 3

- Lesson 5: Component Lifecycle
- Lesson 6: Services and Dependency Injection
- Lesson 7: Angular2 Forms

Day Wise Schedule

➤ Day 4

- Lesson 7: Angular2 Forms (cont.)
- Lesson 8: Internals of Angular 2
- Lesson 9: Routing

References

- <https://angular.io/docs/ts/latest/>
- Ng-book 2 (The complete book on AngularJS 2) by Felipe Coury, Ari Lerner, Nate Murray, & Carlos Taborda



Supplement Technologies

- Bootstrap
- Node Fundamentals

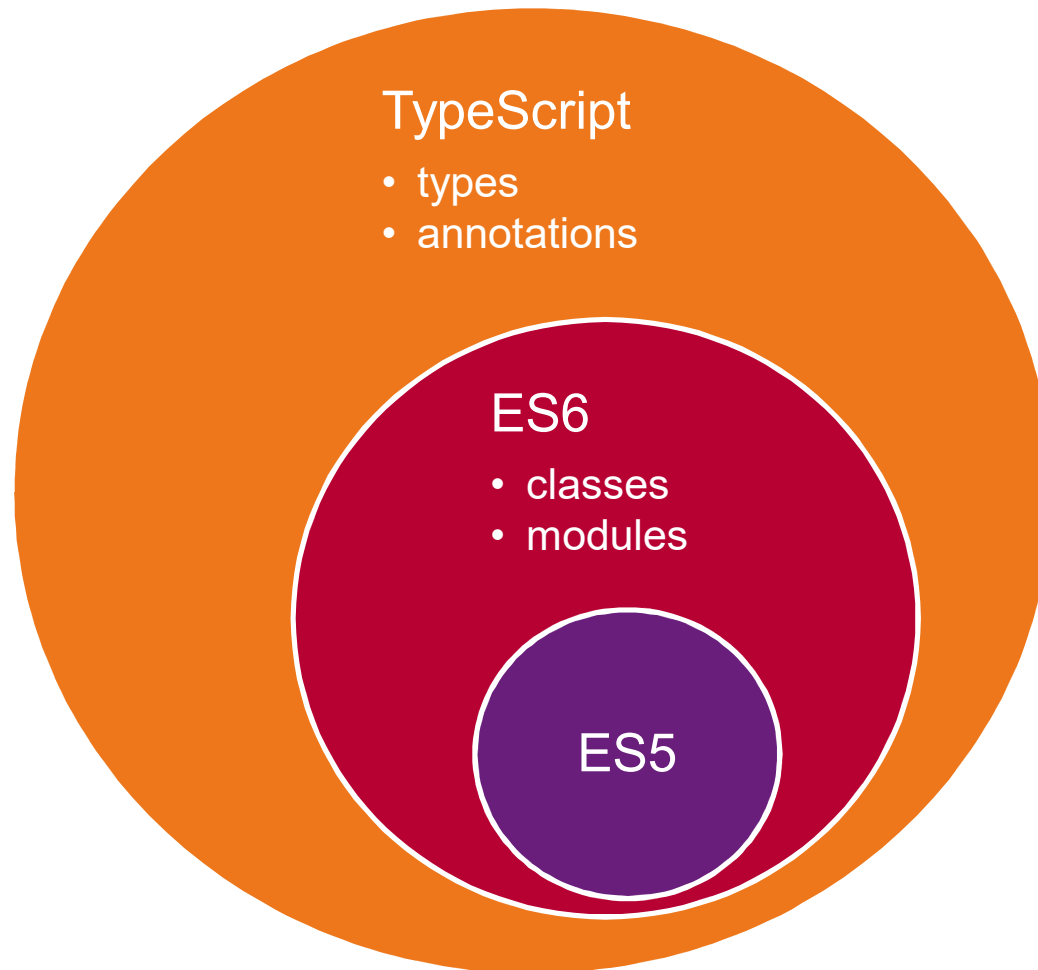
Other Parallel Technology Areas

- React.JS
- Polymer.JS

Angular 2

TypeScript Introduction

TypeScript



ANGULAR 2 IS BUILT IN TYPESCRIPT

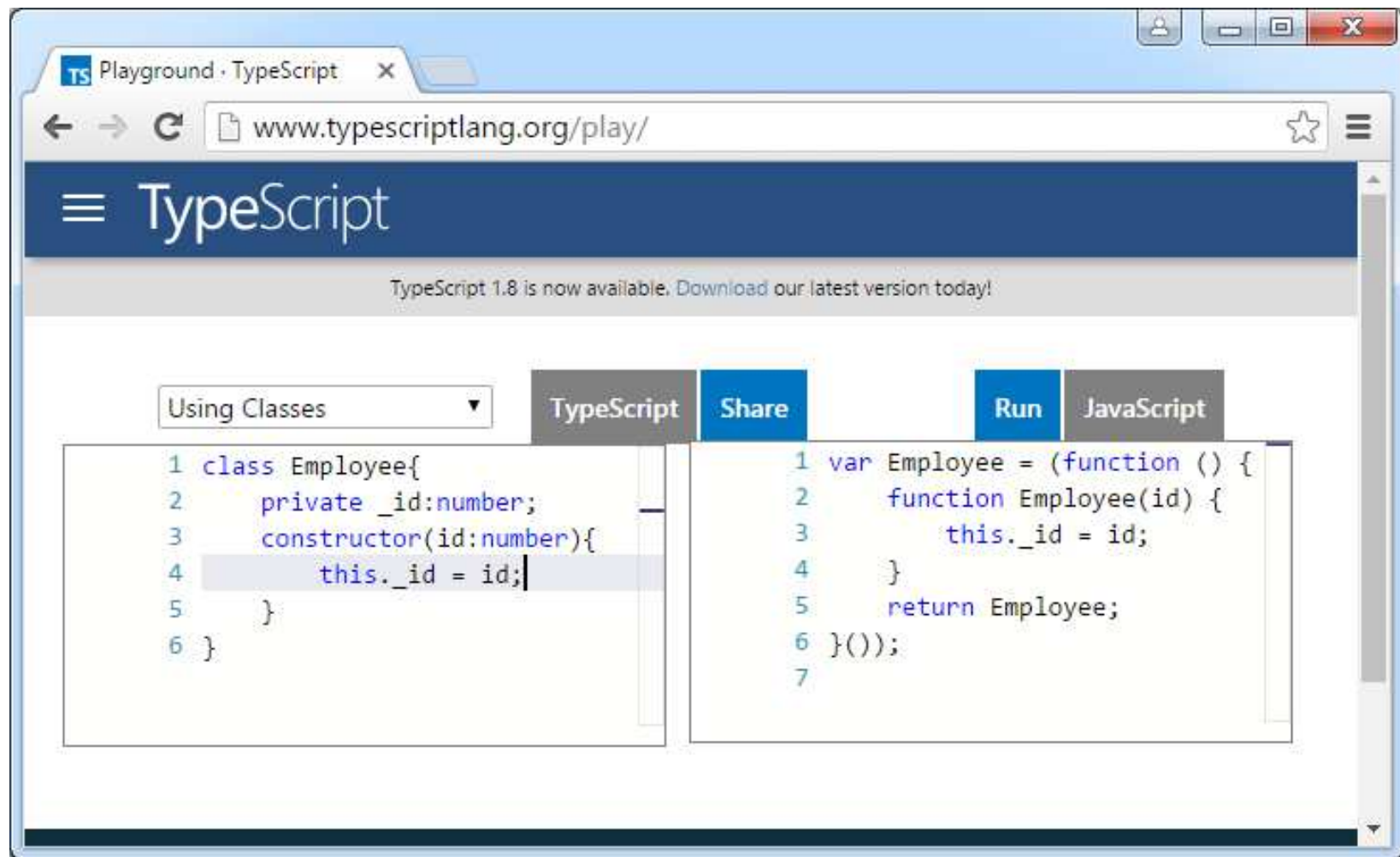
Why TypeScript

- TypeScript can be used for cross browser development and is an open source project
- TypeScript compiles code which has syntax similar to ES6 to simple JavaScript code which runs on any desktop browsers, mobiles, servers / runtimes, or in any other ES3-compatible environment
- Using TypeScript developers can apply class-based approach, compile them down to JavaScript without waiting for the next version of JavaScript.
- With TypeScript existing JavaScript code can be easily incorporated with popular JavaScript libraries like jQuery, Backbone, Angular and so on.
- Enable scalable application development with optional Static types, classes and modules. Static types completely disappear at runtime.
- TypeScript converts JavaScript Programming from loosely typed to strongly typed.

Getting Started with TypeScript

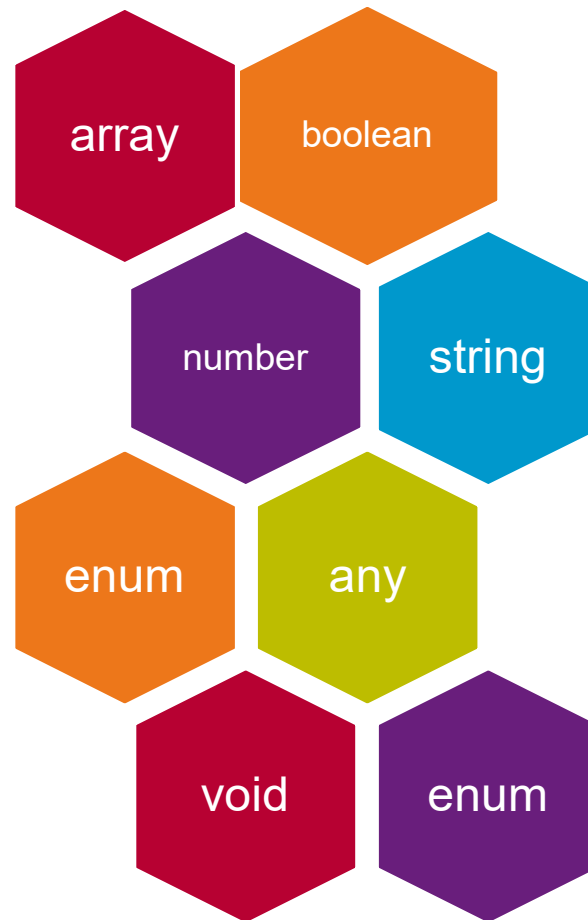
- Visual Studio includes TypeScript in the box, starting with Visual Studio 2013 Update 2 and in Visual Studio 2015.
- Text editors like WebStorm, Atom, SublimeText, Eclipse and Brackets(freeware) can be used.
- TypeScript compiler can be installed for Node Environment by installing it as a node package.
 - `npm install -g typescript` (installing typescript as a global module)
 - `tsc <filename.ts>` (To compile TypeScript from command line).
- In the webpages, we need to refer JavaScript code file (*.js) and not the TypeScript code file (*.ts)
- Microsoft provides an online editor, where we can compile TypeScript Code to Native JavaScript.
 - Link : <http://www.typescriptlang.org/Playground>

TypeScript Playground



Basic Types

- TypeScript supports the following Basic Types



Classes in TypeScript

- Traditional JavaScript focuses on functions and prototype-based inheritance, it is very difficult to build application using object-oriented approach.
- Starting with ECMAScript 6 (the next version of JavaScript), JavaScript programmers can build their applications using this object-oriented class-based approach.
- TypeScript supports public , private and protected access modifiers. Members of a class are public by default.

```
class Employee{  
    private _id:number;  
    constructor(id:number){  
        this._id = id;  
    }  
}
```

Inheritance

- TypeScript allows us to extend existing classes to create new ones using inheritance.
- 'extends' keyword is used to create a subclass.
- 'super()' method is used to call the base constructor inside the subclass constructor.

```
class Foo {  
    constructor() {  
        console.log("Foo constructor Invoked");  
    }  
}  
  
class Baz extends Foo {  
    constructor() {  
        super();  
        console.log("Baz constructor Invoked");  
    }  
}  
  
new Baz();
```

Inheritance

```
class Foo {
    private _privateSample: string;
    public publicSample: string;
    protected protectedSample: string;
    constructor() {
        this._privateSample = "Only in Foo";
        this.publicSample = "Foo-Derived-Instance";
        this.protectedSample = "Foo-Derived";
    }
}
class Baz extends Foo {
    constructor() {
        super();
        this.protectedSample = "Changed Protected in Baz";
        this.publicSample = "Changed Public in Baz";
    }
    print(): void {
        console.log(this.protectedSample);
        console.log(this.publicSample);
    }
}
var bazObj: Baz;
bazObj = new Baz();
bazObj.publicSample = "Changed Public via Instance";
bazObj.print();
```

Accessors

TypeScript supports getters/setters as a way of intercepting accesses to a member of an object.

It gives way of having finer-grained control over how a member is accessed on each object.

```
class Foo {  
    private _name: string;  
    get name():string{  
        return this._name;  
    }  
    set name(name){  
        this._name=name;  
    }  
}  
  
var fooObj = new Foo();  
fooObj.name = "Karthik";  
fooObj.name;
```

Static Property

- In TypeScript we can also create static members of a class, those that are visible on the class itself rather than on the instances.

```
class Foo {  
    static staticVariable:string;  
    instanceVariable:string  
    constructor(instanceVariable:string){  
        this.instanceVariable = instanceVariable;  
    }  
    static staticMethod(){  
        return Foo.staticVariable;  
    }  
}  
  
var fooObj = new Foo("Instance");  
console.log(fooObj.instanceVariable);  
Foo.staticVariable = "Static"  
Foo.staticMethod();
```

Optional, Default & Rest Parameters

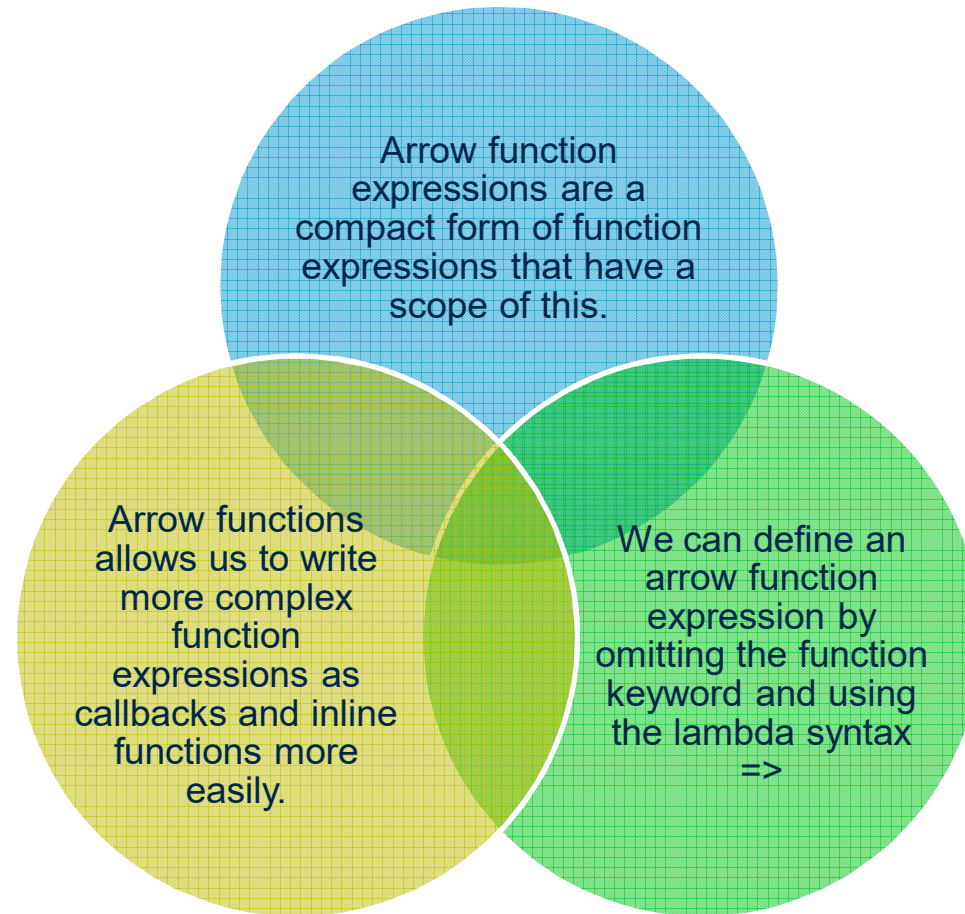
- In JavaScript, every parameter is considered optional, we can get this functionality in TypeScript by using the '?' beside parameters we want optional. Optional parameters must follow required parameters.
 - *function test(firstName:string, lastName?:string){}*
- In TypeScript, we can also set up a value that an optional parameter will have if the user does not provide one. These are called default parameters.
 - *function test(firstName:string, lastName = "Gupta"){}*
- To pass multiple parameters as a group or when we don't know how many parameters a function take we can use parameter followed by (...)
 - *function test(firstName:string, ...others:string[]){}*

Template Strings

- In ES6 new template strings were introduced.
- The two salient features of template strings are
 - Variables within strings (without being forced to concatenate with +)
 - Multi-line strings (using backticks `)

```
let firstName = "Karthik";  
let lastName = "Muthukrishnan";  
// interpolate a string  
let greeting = `Hello ${firstName} ${lastName}`;  
let multiLine = ` Multi  
                  Line  
                  String`;  
console.log(greeting);  
console.log(multiLine);
```

Arrow Functions



```
var addNumbers = (firstNumber:number, secondNumber:number) => firstNumber + secondNumber;  
addNumbers(5, 6);
```

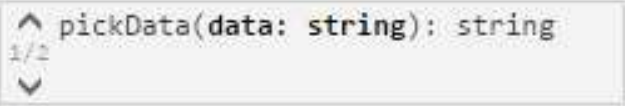

Function Overload

- JavaScript is inherently a very dynamic language.
- JavaScript functions can be overloaded and return different types of objects based on the arguments passed in.

```
function pickData(data:string): string;
function pickData(data:number): number;

function pickData(data): any {
    if (typeof data === "string")
        return data.toUpperCase();
    if (typeof data === "number")
        return data * data;
}

pickData("Karthik");
pickData(5);
```



pickData()

Generics

- Generics plays a vital role in creating reusable components.
- Component can be created to work over a variety of types rather than a single one.

```
function test<T>(data:T):T{  
    return data;  
}  
typeof(test("Karthik"))==="string";//true  
typeof(test(714709))==="number";//true  
  
class Calculator<T>{  
    add:(firstNumber:T,secondNumber:T) => T;  
}  
var calc = new Calculator<number>();  
calc.add = function(x,y){return x+y};  
calc.add(5,6);
```

Interfaces

- Interfaces plays many roles in TypeScript code.
 - Describing an Object : If function take lot of members but you're likely to pass a few of those, we can describe an object using interface with optional parameters
 - Describing an Indexable Object : Using TypeScript interfaces we can represent the expected type of an indexing operation.
 - Ensuring Class Instance Shape: Interface can be used in the same way which we use in traditional OOP languages like C# and JAVA.

```
interface testable{  
    name:string;  
}  
function test(args:testable){  
    console.log(args.name);  
}
```

```
interface CarPart {  
    [name: string]: string;  
}  
class Test{  
    part:CarPart={};  
}  
var testObj = new Test();  
testObj["frame"] = "Frame";
```

```
interface Greetable {  
    greet(message: string): void;  
    language: string;  
}  
  
class Greet implements Greetable{  
    language = 'English';  
    greet(message: string) {  
        console.log(message);  
    }  
}
```

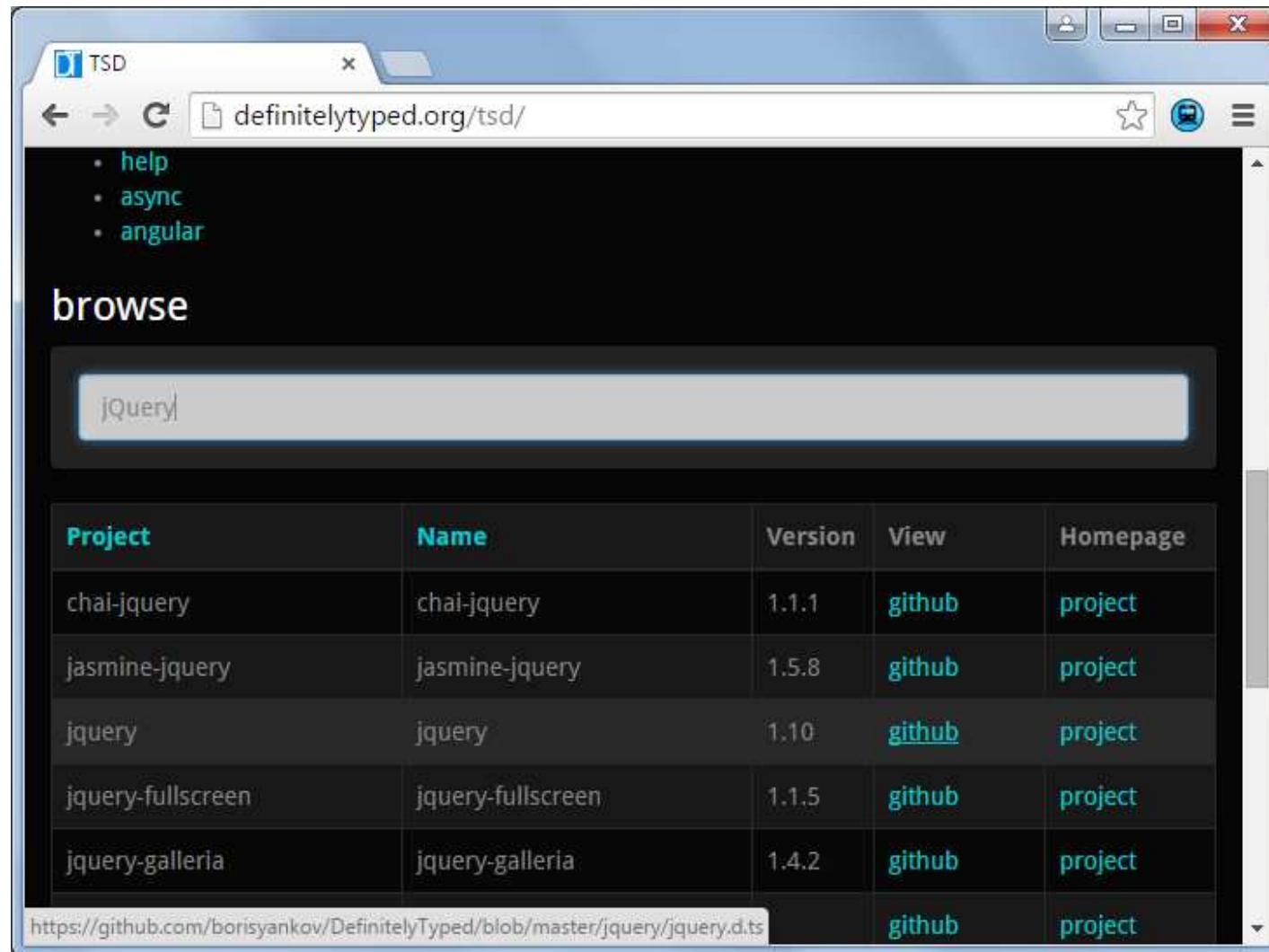
Modules

- A module is the overall container which is used to structure and organize code.
- It avoids collision of the methods/variables with other global APIs.
- TypeScript support modules to organize the code.
- 'export' keyword is used before the classes/ inner module to make it visible outside the module.
- We can refer the multi file internal module using the following line and compile multiple internal file into a single JS file using typescript compiler
 - `/// <reference path="<typescriptfilename>" />`
- External modules can be imported using 'import' keyword
 - `import module1 = require('./Module1');`

Declaration file (.d.ts)

- Documentation helps developers to understand the implementation details of JavaScript libraries.
- It is very much important to describe the shape of an external JavaScript library or new host API, so that it will make developer's life easier rather reading and understanding the JS code.
- Using a declaration file (.d.ts) in TypeScript, we can describe the shape and provide the documentation for the JavaScript library.
 - To generate declaration file use the command : `tsc --declaration <tsfilename>`
- definitelytyped.org is the repository for high quality TypeScript type definitions for popular JavaScript Libraries like jQuery, Angular, Backbone etc.

definitelytyped.org



Person.ts

```
class Person {
    private _age: number;
    private _name: string;

    constructor(name?: string, age?: number) {
        this._age = age;
        this._name = name;
    }

    get age(): number {
        return this._age;
    }

    get name(): string {
        return this._name;
    }

    getPersonInformation(): string {
        return "Name : " + this._name + " Age: " + this._age;
    }
}
```

Person.d.ts

```
declare class Person {
  private _age;
  private _name;
  /**
   * Default constructor to create a person Instance
   */
  constructor();
  /**
   * constructor takes name and age for a person Instance
   * @param name string type used to assign the name for the person
   * @param age number type used to assign age for the person
   */
  constructor(name: string, age: number);
  /**
   * get the age for person
   */
  age: number;
  /**
   * get the name for person
   */
  name: string;
  /**
   * get the person information
   * returns the name and age as string
   */
  getPersonInformation(): string;
}
```


app.ts

```
/// <reference path="person.d.ts" />  
var p = new Person("Ganesh",
```

▲ 2 of 2 ▼ Person(name: string, age: number): Person
constructor takes name and age for a person Instance
age: number type used to assign age for the person

Summary

- TypeScript is an open source project maintained by Microsoft.
- TypeScript generates plain JavaScript code which can be used with any browser.
- TypeScript offers many features of object oriented programming languages such as classes, interfaces, inheritance, overloading and modules, some of which are proposed features of ECMA Script 6.
- TypeScript is a promising language that can certainly help in writing neat code and organize JavaScript code making it more maintainable and extensible.
- Angular 2 is built in typescript



Angular 2

Angular 2 Fundamentals

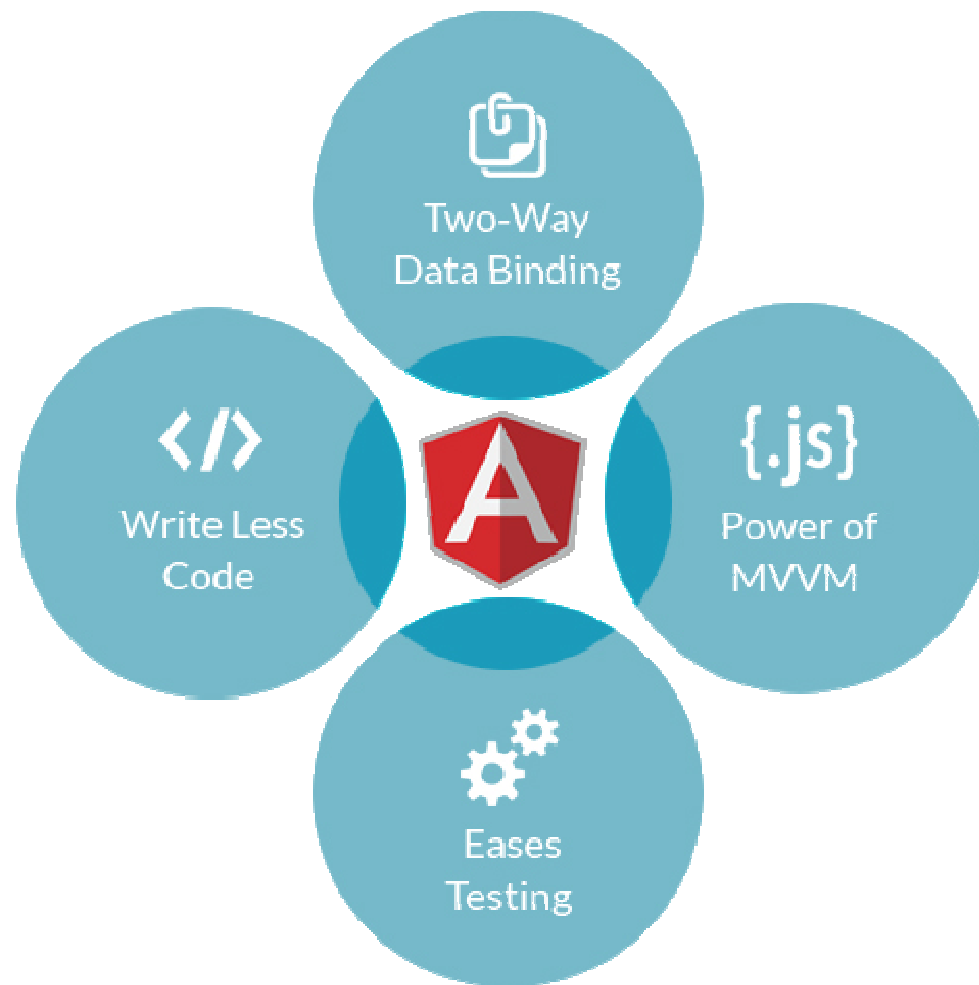
Angular 2 Introduction

- Angular 2 is the next version of Google's massively popular MV* framework.
- Angular 2 is a framework to help us build client applications in HTML and either JavaScript or a language (like Dart or TypeScript) that compiles to JavaScript.
- Angular 2 comes with almost everything required to build a complicated frontend web or mobile apps, from powerful templates to fast rendering, data management, HTTP services, form handling, and so much more.

Why Angular

- Angular makes HTML more expressive, It powers up HTML with features such as if conditions, for loops and local variables.
- Angular has powerful data binding. We can easily display fields from our data model, track changes and process updates from the user.
- Angular promotes modularity by design so that the applications become a set of building blocks making it easier to create and reuse contents.
- Angular has built-in support for communication with a backend service this makes it easy for Web applications to integrate with the backend service to GET and POST data or execute server side business logic.

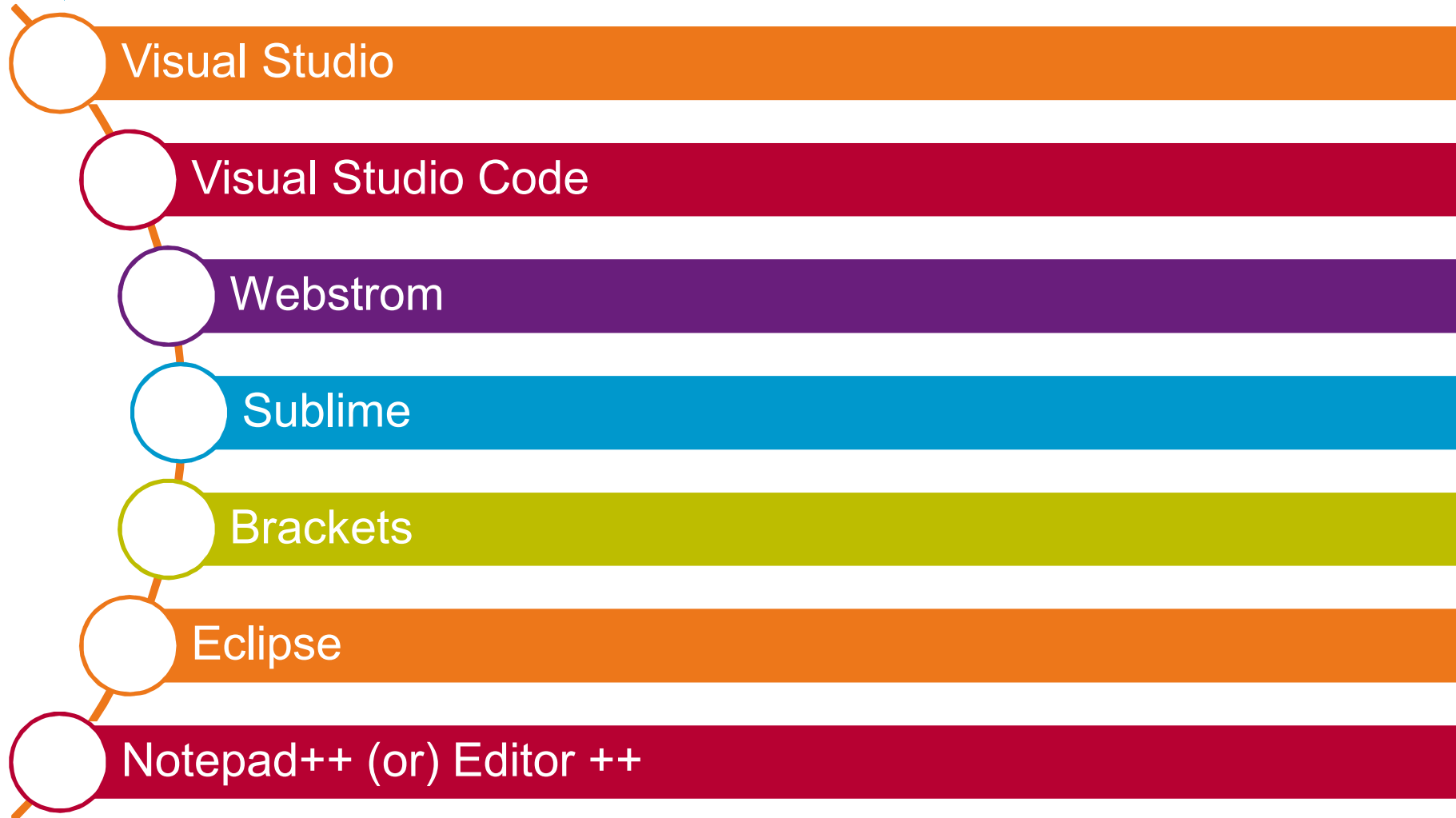
Why Angular



Why Angular 2

- Angular 2 was built for speed, It has faster initial loads faster change detection and improved rendering times.
- Angular 2 is modern it takes advantage of features provided in the latest JavaScript standards such as classes, modules and decorators.
- It leverages web component technologies for building reusable user interface widgets.
- It supports both modern and legacy browsers like Chrome, Firefox and Internet Explorer back to IE 9.
- It has a simplified API. It has fewer built-in directives to learn simple binding and a lower overall concept count.
- It enhances productivity to improve day to day workflows

Angular 2 Code Editors



Language Choice

ES5

ES6

TypeScript

Dart

Angular 2 Dependencies

- To run Angular 2, we depend on these four libraries:.

core-js

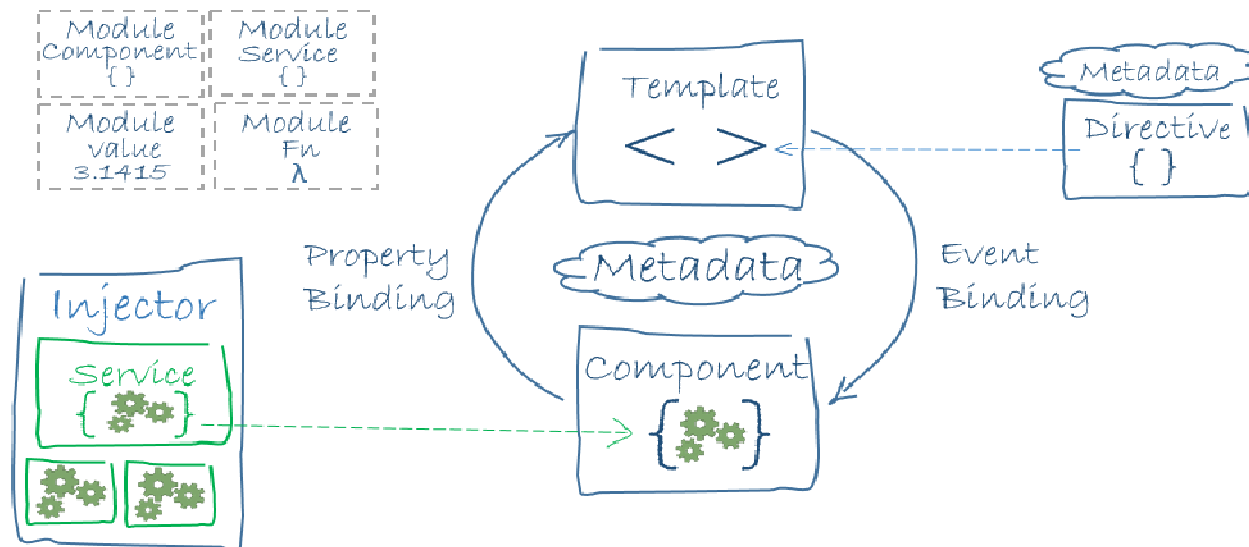
zone.js

reflect-metadata

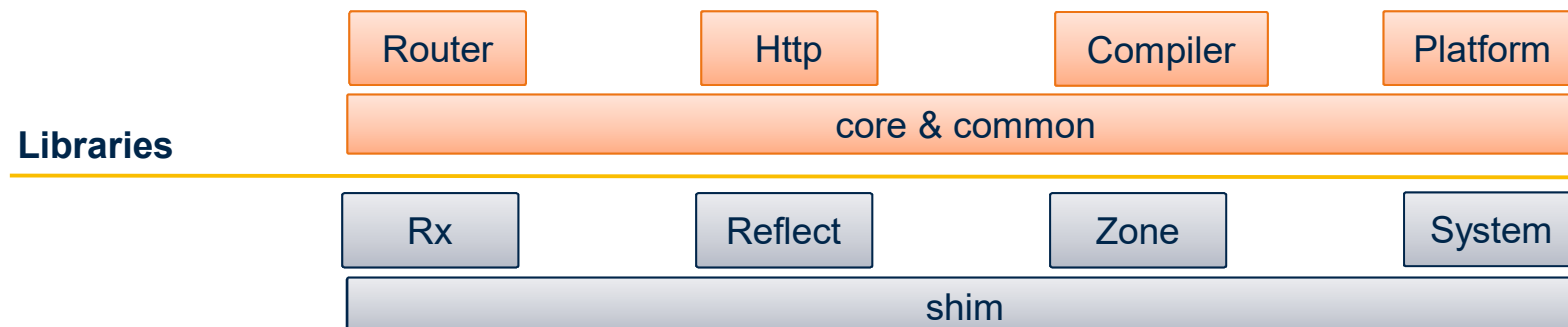
SystemJS

Building Blocks of an Angular 2

Application

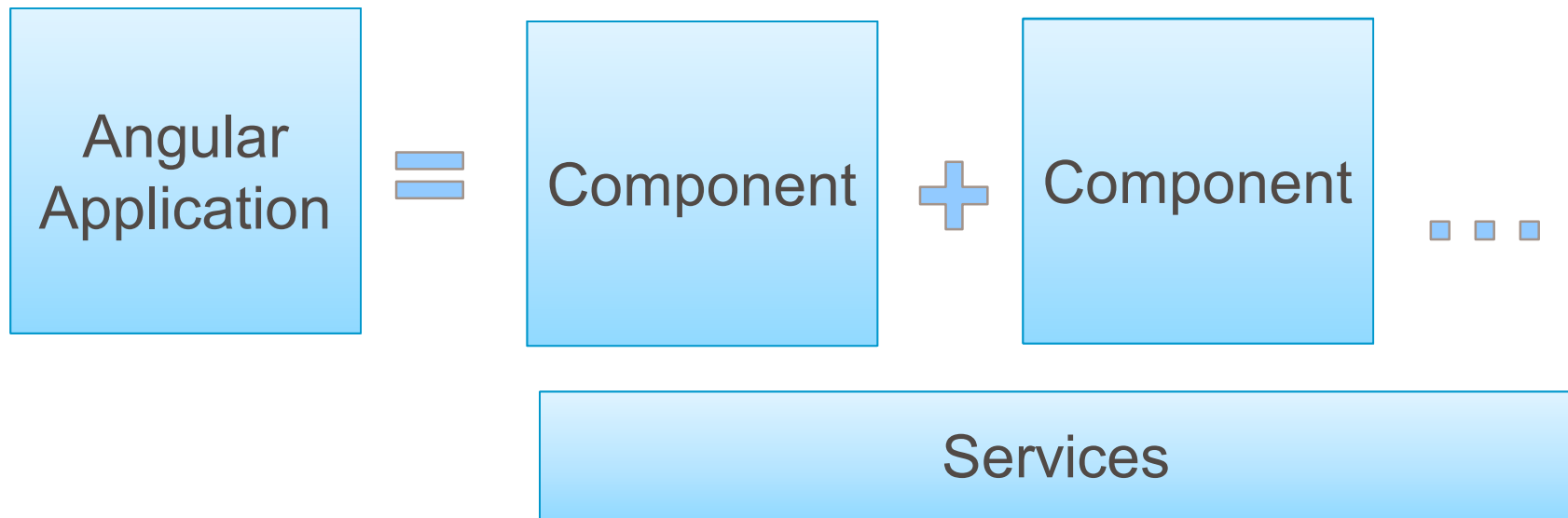


Angular Frameworks



Angular 2 Application

- Angular 2 application is comprised of a set of components and some services that provide functionality across those components.



Importing Modules

- Module loader finds an external function or class using the *import* statement.
- *imports* statement allows us to use exported members from external modules. External modules can be a third party library or our own modules or from angular itself.
- If multiple members from the same module is needed, It can be listed in the import list separated by commas.

```
import { NgModule } from '@angular/core'
```

- Angular is a collection of library modules, each library is itself a module made up of several related feature modules.



Root Module

- An Angular module class describes how the application parts fit together.
- Every application has at least one Angular module, the root module that you bootstrap to launch the application.
 - The conventional name for the root module is AppModule .

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})

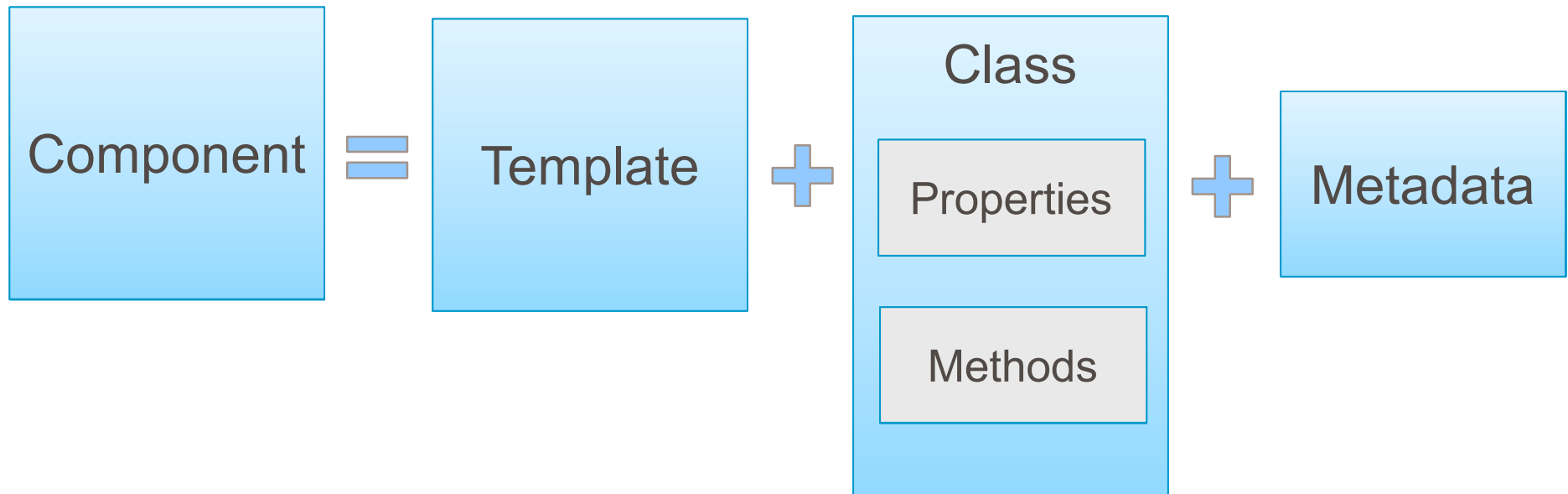
export class AppModule { }
```

Introduction to Angular Components

- Components are the main way to build and specify elements and logic on the page.
- Component is comprised of a template and class.
 - Template provides HTML(View) for the user interface.
 - Class provides the code associated with the view.
 - Class contains the properties or data elements to be used in the view and methods to perform actions for the view.
- Component also has metadata, which provides additional information about the component
 - Meta data that identifies the class as an angular component.

Angular 2 Component

- A component contains application logic that controls a region of the user interface (view)



Angular Component Using TypeScript

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

```
@Component({  
  selector: 'my-app',  
  template: `<div>  
    <h2>{{greet}}</h2>  
  </div>`  
})
```



Metadata
&
Template

```
export class AppComponent {  
  greet:string = "Hello Angular2!"  
}
```



Class

Component Class

- Class is a construct that allows to create a type with properties and methods.
- As per convention, name the component class with a feature name then append the word *component* as the suffix.
- Also by convention root component for an application is called AppComponent.
- Class name is used as the component name when the component is referenced in code.
- export keyword exports the class; thereby making it available for use by other components of the application.

```
export class AppComponent {  
    greet:string = "Hello Angular2!"  
}
```

Component Metadata

- A class becomes an angular component, when component meta data is given. Metadata tells Angular how to process a class.
- Angular needs metadata to understand, how to instantiate the component, construct the view and interact with the component.
- Components meta data is defined with the angular *Component* function. In Typescript this function is attached to the class as a decorator.
- *@Component* decorator is used to identify the class as a component. It takes object as a parameter which has many properties like selector, template etc
 - selector defines the components directive name which is used to reference the component in HTML.
 - Whenever this directive is used in the HTML angular renders this components template

Angular 2 Metadata

- The main objective @Component decorators is to add meta-data to the application which tells Angular 2 how to process a class.
- When building Angular components we can configure the following options:
 - **selector** : defines the name of the HTML tag where the component resides
 - **providers**: Used to pass services for the component
 - **styles** : Used to style a specific component.
 - **template** : This is the portion of our component that holds template. It is an integral part of the component as it allows to tie logic from component directly to a view. It can be either inline, or external template using **templateUrl** to link to an external template.

Demo

- Creating-AppComponent



Template

- Template are mostly HTML which is used to tell Angular how to render the component.
- Template for a component can be created using
 - Inline template (Embedded template string)
 - Linked template (Template provided in external html file)
- Inline template can be defined with ***template*** property using a single or double quotes or with a multiline string by enclosing the HTML in ES 2015 back ticks.
 - Back ticks allows to compose multiline string which makes the HTML more readable.
- Linked template is used to define the HTML in its own file by providing the URL of the HTML file using ***templateUrl*** property.
 - Path provided in templateUrl property is relative to the application root which is usually the location of the index.html

Demo

- InlineTemplate
- LinkedTemplate



Component Styles

- Angular 2 applications are styled with regular CSS. i.e. CSS stylesheets, selectors, rules, and media queries can be directly applied.
- Angular 2 has the ability to encapsulate component styles with components enables more modular design than regular stylesheets.
- In Angular 2 component, CSS styles can be defined like HTML template in several ways
 - As inline style in the template HTML
 - Template Link Tags
 - By setting ***styles*** or ***styleUrls*** metadata

Demo

- ComponentStyles



Summary

- Every application must bootstrap at least one component, the root application component.
- As a best practice, We need to launch the application by bootstrapping the AppModule in the main.ts file.
- The bootstrap array should be used only in root application module i.e. AppModule
- NgModule is a way to organize the dependencies for compiler and dependency injection.
- NgModule can import other modules as dependencies
- Angular uses a library called **Zone.js** to automatically detect the things changed and trigger the change detection mechanism



Summary

- Every component must be declared in some NgModule and a component can belong to one and only one NgModule
- exports key is nothing but the list of public components for NgModule.
- Angular2 Application is a tree of components and the top level component is nothing but the application.
- Components are Composable.
- Template for a component can be created using InlineTemplate and LinkedTemplate using template and templateUrl respectively.
- **styles** and **styleUrls** keys are used in components to work with styles in Angular 2

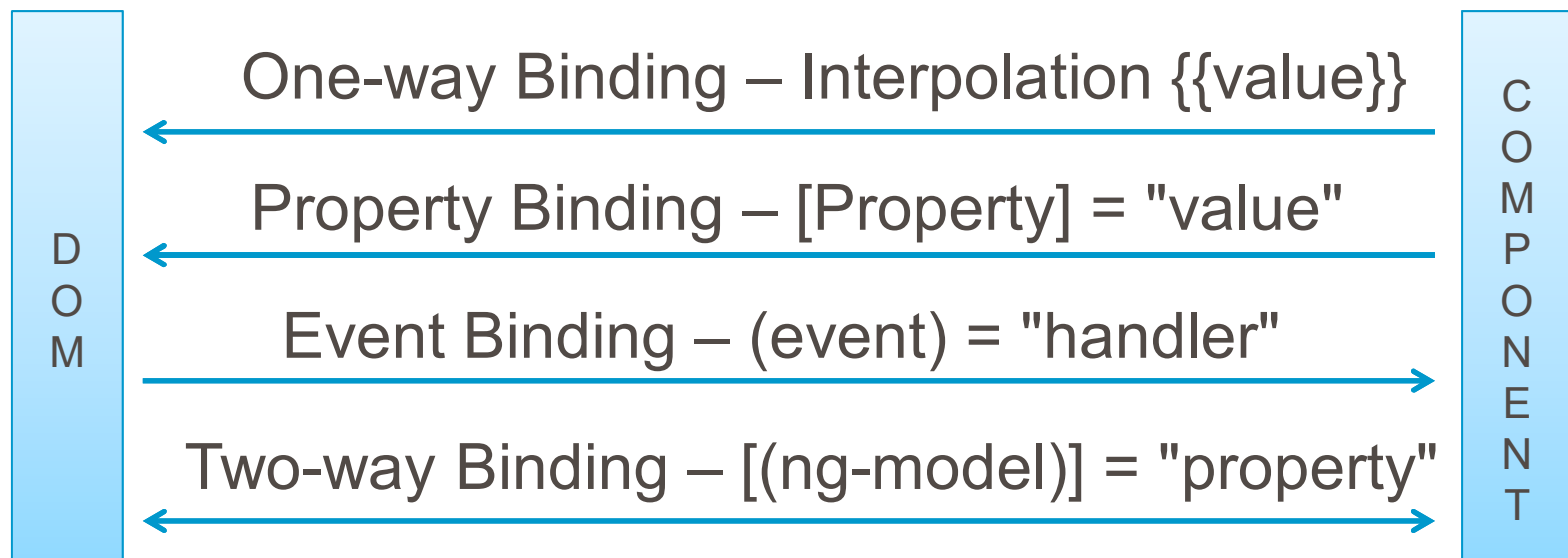


Angular 2

Data Binding

Databinding

- Angular supports data binding, a mechanism for coordinating parts of a template with parts of a component.
- Binding markups are added to the template HTML to tell Angular how to connect both sides.
- There are four forms of data binding syntax. Each form has a direction to the DOM, from the DOM, or in both directions.



Interpolation(One-Way binding)


- Interpolation performs one way binding from the class property to the template given in a double curly braces.
- Using Interpolation operations like concatenation, simple calculations, calling method on a class can be performed.
- Interpolation is used to insert the interpolated strings into the text between HTML elements.
- The syntax between the interpolation curly braces is called as template expression.
- Angular evaluates that expression using the component as the context.
 - Angular looks to the component to obtain property values or to call methods.
 - Angular then converts the result of the template expression to a string and assigned that string to an element or directive property


```
<h2>{{greet}}</h2>    {{'Hi' + greet}}    {{'Hi' + getMessage()}}    {{2+1}}
```

Property Binding

- Property binding allows us to set a property of an element to the value of a template expression.

```
<h1 [innerText]='msg'></h1>
```


BindingTarget


Binding Source

- Binding target is always enclosed in square brackets and the Binding source is always enclosed in quotes and specifies the template expression
- Like interpolation property binding is one way from the source class property to the target element property
- Property binding effectively allows to control the template DOM from a component class.
- The general guideline is to prefer property binding all for interpolation. However to include the template expression as part of a larger expression then use interpolation

Demo

- PropertyBinding



Event Binding

- Event binding is used to send information from the element property to the component class property to respond for user events.
 - For instance to perform an operation when the user clicks a button a component listens for user actions using event binding.

<pre><button (click)='sayHi()'>SayHi</button></pre> <div><div>TargetEvent</div><div>TemplateStatement</div></div>

- The name of the bound event is enclosed in parentheses identifying it as the target event.
- Template statement is often the name of a component class method enclosed in quotes.
- If the defined event occurs; the template statement is executed calling the specified method in the component

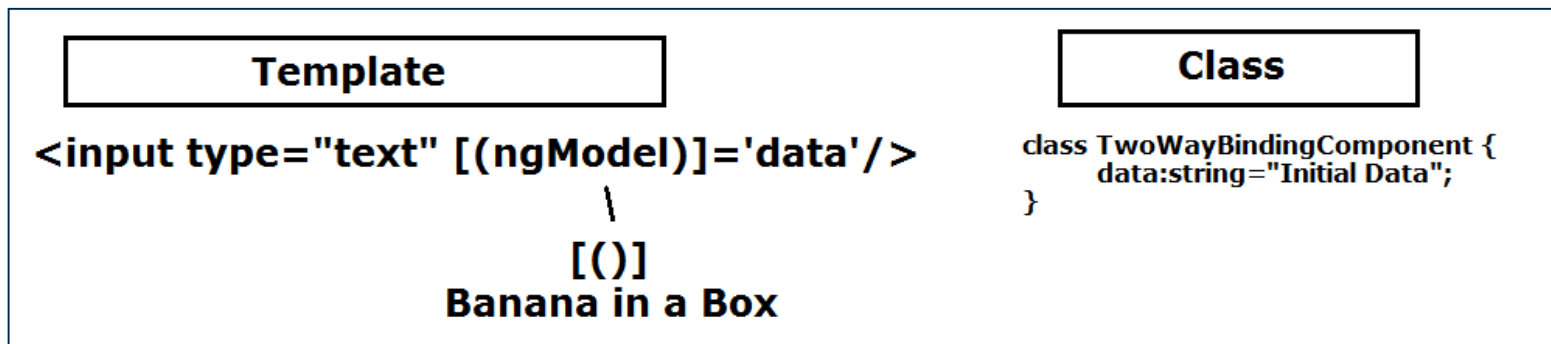
Demo

- EventBinding
- GetUserInput



Two-way Binding

- To display a component class property in the template and update that property when the user makes a change with user entry HTML elements like input element two-way binding is required.
- In Angular ***ngModel*** directive is used to specify the two way binding.



- `ngModel` in square brackets is used to indicate property binding from the class property to the input element
- Parentheses to indicate event binding to send the notification of the user entered data back to the class property

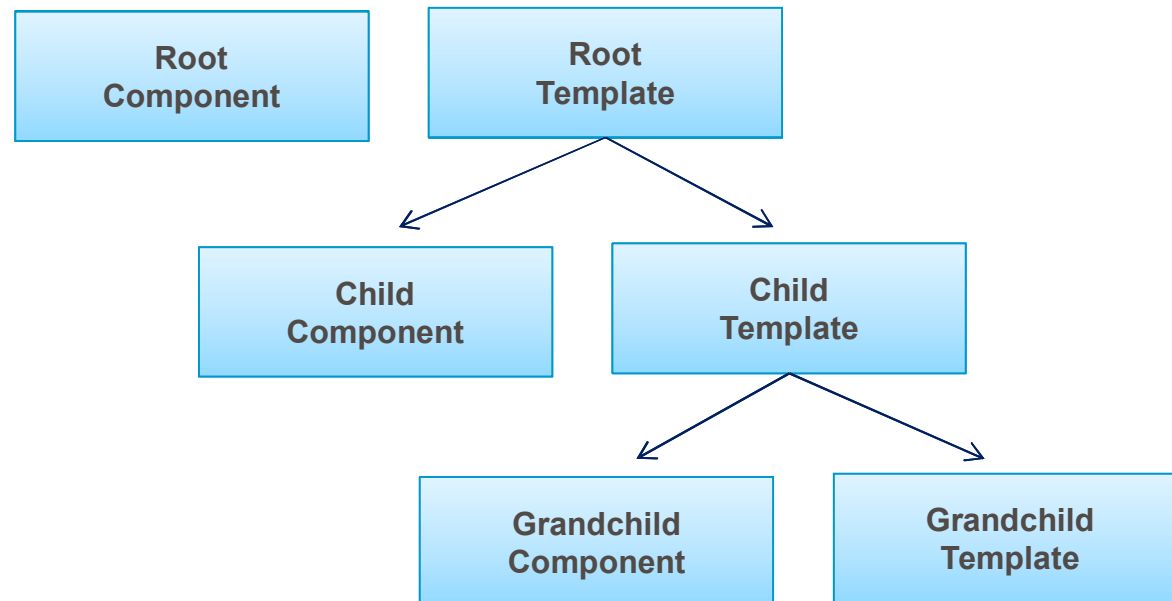
Demo

- TwoWayBinding



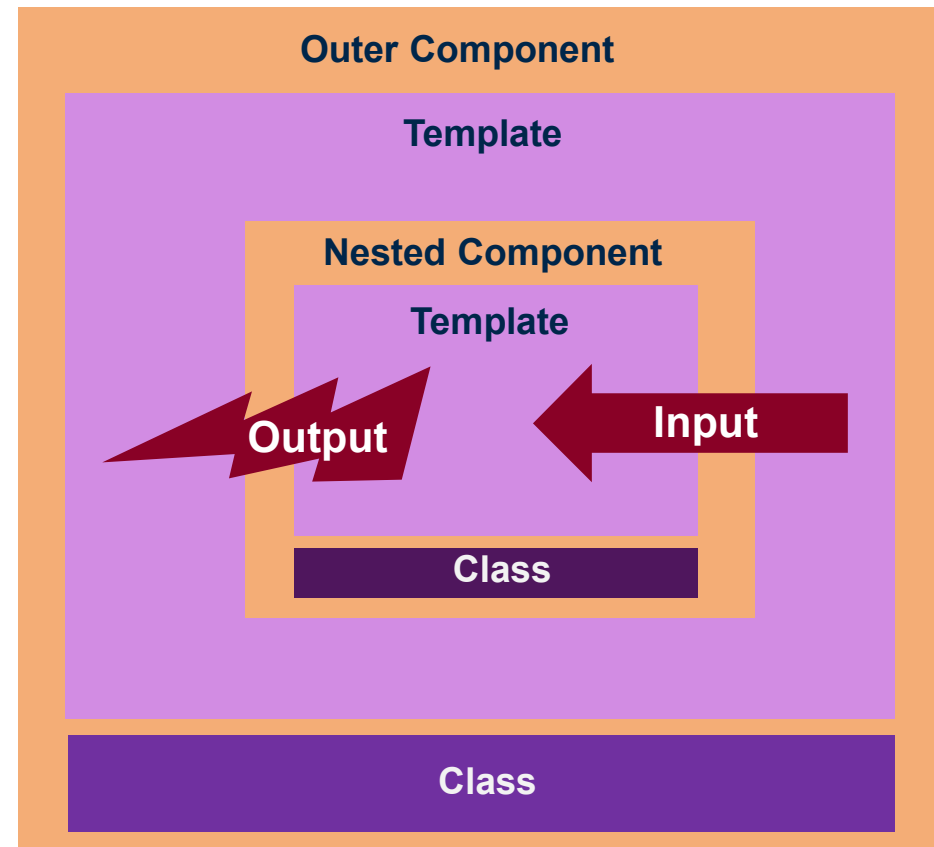
Nested Components

- Components have templates which may contain other components.
- Outer component is referred as the container or parent component
- Inner component is referred as the nested or child component.



Using @Input and @Output

- Nested component receives information from its container using input properties(@Input) and outputs information back to its container by raising events(@Output).
- Input, Output & EventEmitter need to be imported in Nested component from @angular/core module.
- emit() method is used to trigger the event by emitting data from inner component to outer component which can be accessed via \$event



Demo

- Building-NestedComponents



Summary

- Angular provides 4 forms of data binding syntax
 - One-way Binding – Interpolation {{value}}
 - Property Binding – [Property] = "value"
 - Event Binding – (event) = "handler"
 - Two-way Binding – [(ng-model)] = "property"
- Nested component receives information from its container using input properties(@Input) and outputs information back to its container by raising events(@Output).
- When Parent component render, it recursively render its children component
- emit() method is used to trigger the event by emitting data from inner component to outer component which can be accessed via *\$event*



Angular 2

Directives and Pipes

Directives

- Directives power up the HTML.
- It is used to attach behavior to elements in the DOM
- There are three kinds of directives in Angular 2



Component Directives

- Directive with a template

Structural Directives

- Directive used to change the DOM Layout

Attribute Directives

- Directive used to change the appearance or behavior of an element

Demo

- ComponentDirective



Structural Directives

- Structural directives modifies the structure or layout of a view by adding, removing or manipulating elements and their children.
- '*' in front of the directive name marks the directive as a structural directive.
- In angular we have three built-in structural directives
 - **ngIf** : ngIf directive inserts or removes an element based on a truthy/falsey condition.
 - **ngFor** : ngFor directive is used to iterate an array of items
 - **ngSwitch**: ngSwitch directive is used to conditionally swap DOM structure on template based on an expression.

Demo

- ngIf-Directive
- ngFor-Directive
- ngSwitch-Directive



Attribute Directives

- Attribute directives alter the appearance or behavior of an existing element.
- In templates they look like regular HTML attributes.
- Some important angular in-built attribute directives are :
 - **ngModel** : Implements two-way data binding, which modifies the behavior of an existing element (typically an `<input>`) by setting its display value property and responding to change events.
 - **ngStyle** : Changes the style based on a result of expression evaluation.
 - **ngClass** : Conditionally adds and removes CSS classes on an HTML element based on an expression's evaluation result

Demo

- NgStyle-Directive
- NgClass-Directive



@HostBinding and @HostListener

- Host property decorators are used to bind a host element to a component or directive.
- @HostBinding is used to bind the host element property to a directive property. Angular automatically checks host property bindings during change detection. If a binding changes, it will update the host element of the directive.
- @HostListener will listen to the event emitted by host element, declared with @HostListener.

Demo

- CustomDirective



Pipe

- Pipes are used to transform displayed values within a template.
- Pipes transform bound properties before they are displayed
- A pipe takes in data as input and transforms it to a desired output.
- To pass an argument to a pipe in the HTML form, pass it with a colon after the pipe (for multiple arguments, simply append a colon after each argument)
- Angular gives us several built-in pipe like lowercase, date, number, decimal, percent, currency, json, slice etc
- Angular provides a way to create custom pipes as well.

Built-in Pipes

- Format Pipes
 - DatePipe
 - UpperCasePipe
 - LowerCasePipe
 - CurrencyPipe
 - PercentPipe
 - JsonPipe
- Array pipes
 - SlicePipe
- Async pipes
 - AsyncPipe

Custom Pipes

- A pipe is a class decorated with pipe metadata
- The pipe class implements a transform method
 - Takes an input value and an optional array of parameter strings
 - Returns the transformed value

```
import {PipeTransform,Pipe} from '@angular/core';

@Pipe({ name : 'customPipe'})

export class ExponentialStengthPipe implements PipeTransform{
    transform(value:number,args:string[]):any {
        return Math.pow(value,parseInt(args[0] || '1', 10));
    }
}
```

Demo

- WorkingWithPipes
- CustomPipe



Summary

- Component Directives is a directive with a template.
- Directives can't be bootstrapped
- Structural directives change the DOM layout by adding and removing DOM elements.
- Attribute directives changes the appearance or behavior of a DOM element.
- In order to use *ngModel* in the application components, we need to compulsorily add FormsModule in the Imports array of Application Module class
- *ngNonBindable* tells the Angular not to compile or bind a particular section of a DOM.
- Using *ngStyle* directive we can set CSS properties for the DOM element from Angular expressions



Summary

- *ngClass* directive allows us to dynamically set and change the CSS classes for a given DOM element
- Pipes are used to transform displayed values within a template.
- The pipe class implements a transform method which takes an input value and an optional array of parameter strings which returns the transformed value

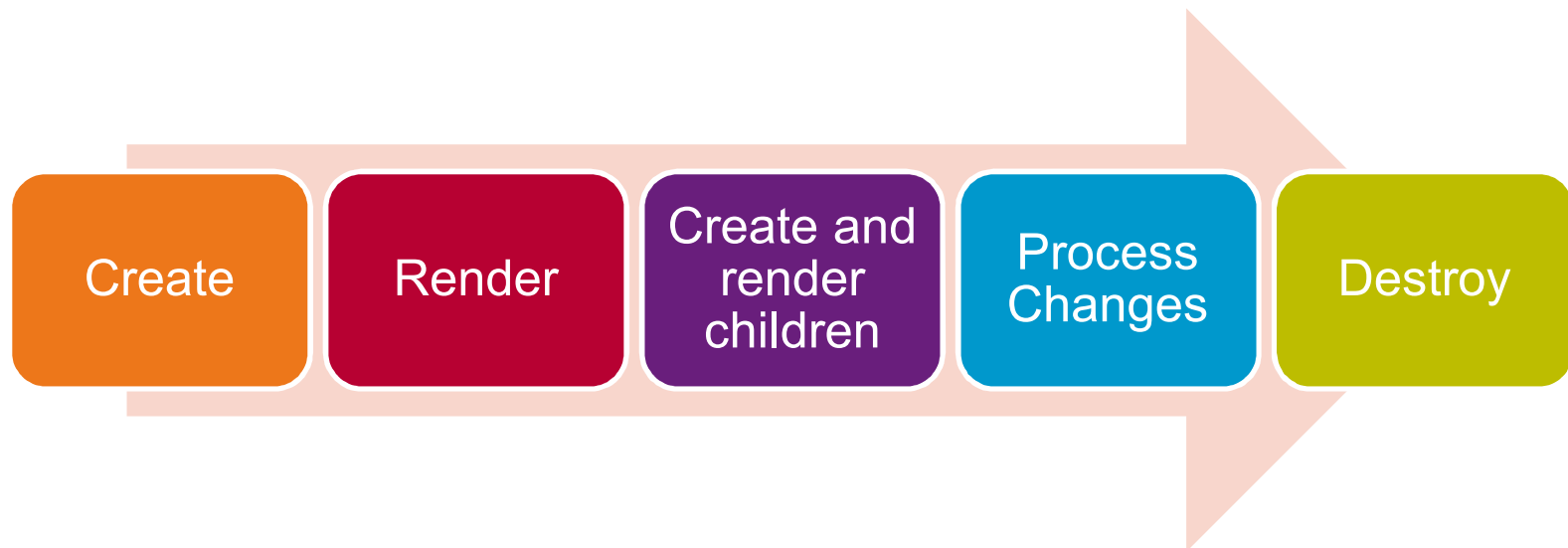


Angular 2

Component Lifecycle

Component Lifecycle

- A component has a lifecycle managed by angular.
- The constructor of the component class is called before any other component lifecycle hook.
- As a best practice inputs of a component should not be accessed via constructor. To access the value of an input for instance to load data from server component's life cycle phase should be used



Lifecycle hooks

- Angular calls life cycle hook methods on directives and components as it creates, changes and destroy them

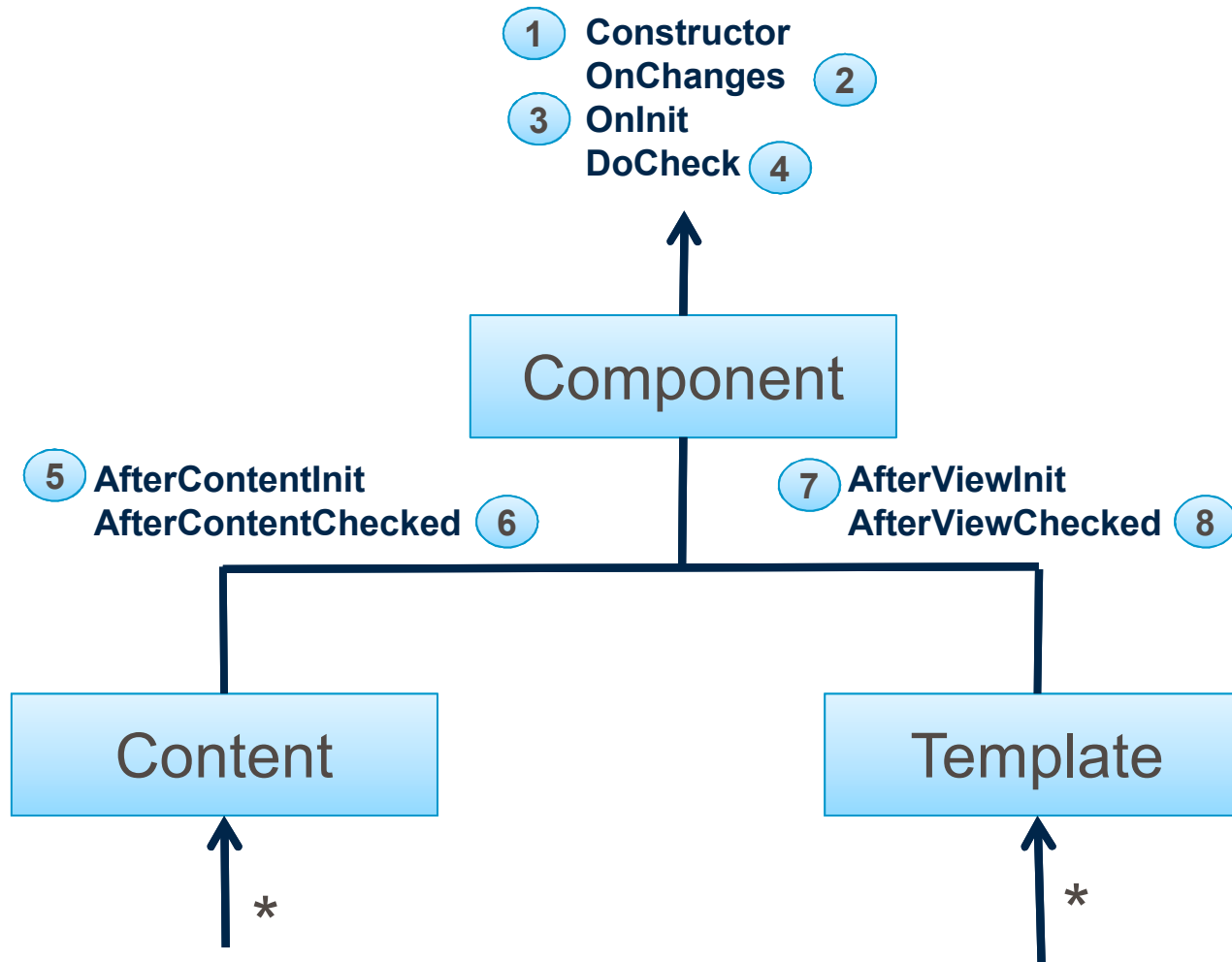
Creates	Changes	Destroy
OnInit	DoCheck	OnDestroy
AfterContentInit	OnChanges	
AfterViewInit	AfterContentChecked	
	AfterViewChecked	

Accessing Component Lifecycle hooks

- To respond to certain events in the life cycle of a component angular provides various interface which can be implemented.
- Angular application finds the hook functions only when the Component class assigns the associated interface.
- For each hook a corresponding interface exists without the prefix **ng**

Interface	Hook	Interface	Hook
OnChanges	ngOnChanges	OnInit	ngOnInit
DoCheck	ngDoCheck	AfterContentInit	ngAfterContentInit
AfterContentChecked	ngAfterContentChecked	AfterViewInit	ngAfterViewInit
AfterViewChecked	ngAfterViewChecked	OnDestroy	ngOnDestroy

Lifecycle hooks execution order



Demo

- ComponentLifeCycle



Summary

- Angular calls lifecycle hook methods on directives and components as it creates, changes, and destroys them.
- After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods.
- An `ngOnInit` is a good place for a component to fetch its initial data.
- Angular calls its `ngOnChanges` method whenever it detects changes to input properties of the component (or directive).
- For each hook a corresponding interface exists without the prefix ***ng***



Angular 2

Services and Dependency Injection

Dependency Injection

- Dependency injection is a design pattern that allows for the removal of hard-coded dependencies, thus making it possible to remove or change them at run time.

```
function Foo(object) {  
    this.object = object;  
}  
  
Foo.prototype.showDetails = function(data) {  
    this.object.display(data);  
}  
  
var greeter = {  
    display : function(msg){  
        alert(msg);  
    }  
}  
  
var foo = new Foo(greeter);  
foo.showDetails("Capgemini");
```


Dependency Injection in Angular 2

- DI allows to inject dependencies in different components across applications, without needing to know, how those dependencies are created, or what dependencies they need themselves.
- DI can also be considered as framework which helps us out in maintaining assembling dependencies for bigger applications.
- Angular 1 has it's own DI system which allows us to annotate services and other components and let the injector find out, what dependencies need to be instantiated. But it has the following limitations.
 - Internal Cache
 - Namespace Collision
 - Built into the framework

Services

- Services provided architectural way to encapsulate business logic in a reusable fashion.
- Services allow to keep logic out of your components, directives and pipe classes
- Services can be injected in the application using Angular's dependency injection (DI).
- Angular has In-built service classes like Http, FormBuilder and Router which contains logic for doing specific things that are non component specific.
- Custom Services are most often used to create Data Services.

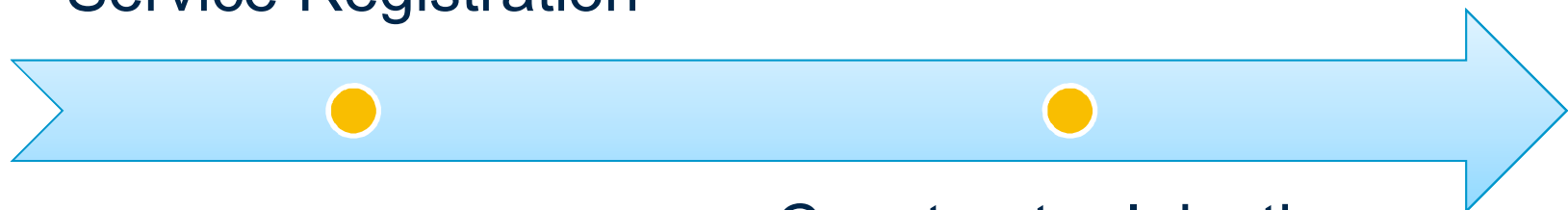
Working with Services in Angular 2

- Component can work with service class using two ways
 - Creating an instance of the service class
 - Instances are local to the component, so data or other resources cannot be shared
 - Difficult to test the service
 - Registering the service with angular using angular Injector
 - Angular injector maintains a container of created service instances
 - The injector creates and manages the single instance or singleton of each registered service.
 - Angular injector provides or injects the service class instance when the component class is instantiated. This process is called dependency injection
 - Angular manages the single instance any data or logic in that instance is shared by all of the classes that use it. This technique is the recommended way to use services because it provides better management of service instances it allow sharing of data and other resources and it's easier to mock the services for testing purposes

Working with Services in Angular 2

- Angular has dependency injection support baked into the framework which allows to create component directives in modular fashion.
- DI creates instances of objects and inject them into places where they are needed in a two step process.

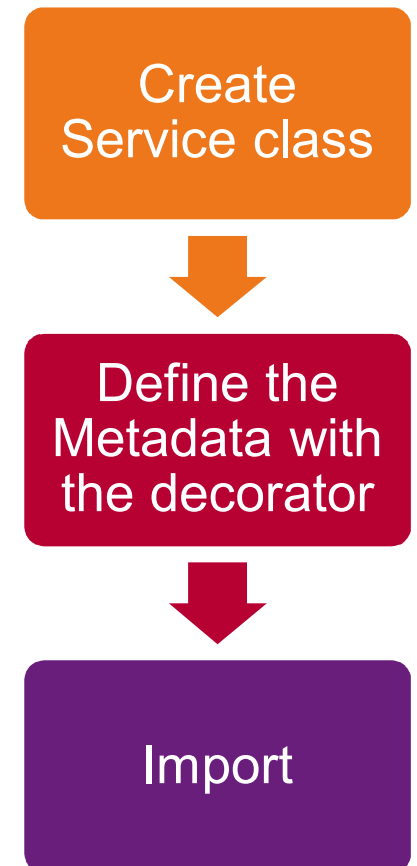
Service Registration



Constructor Injection

Building a Service

- Steps to build a service is similar to build components and a custom pipe.
- It is recommended that every service class use the injectable decorator for clarity and consistency.
- **@Injectable** is a decorator, that informs Angular 2 that the service has some dependencies itself. Basically services in Angular 2 are simple classes with the decorator @Injectable on top of the class, that provides a method to return some items.



Registering a Service

- To register a service we must register a provider.
- A provider is code that can create or return a service typically the service class itself.
- To register a provider define it as part of the component Metadata, so that Angular injector can inject the service into the component and any of its children.

Injecting a Service

- In typescript a constructor is defined with a constructor function.
- The constructor function is executed when the component is created.
 - It is primarily use for initialization and not for the code that has side effects or takes time to execute.
- Dependency can be injected as a parameters to the constructor function.
 - When class is constructed the angular injector sets this parameter to the injected instance of the requested service.
 - The injected service instance can be assigned to the local variable by adding the private keyword to the constructor parameter, so that it can be accessed anywhere in the class.

Demo

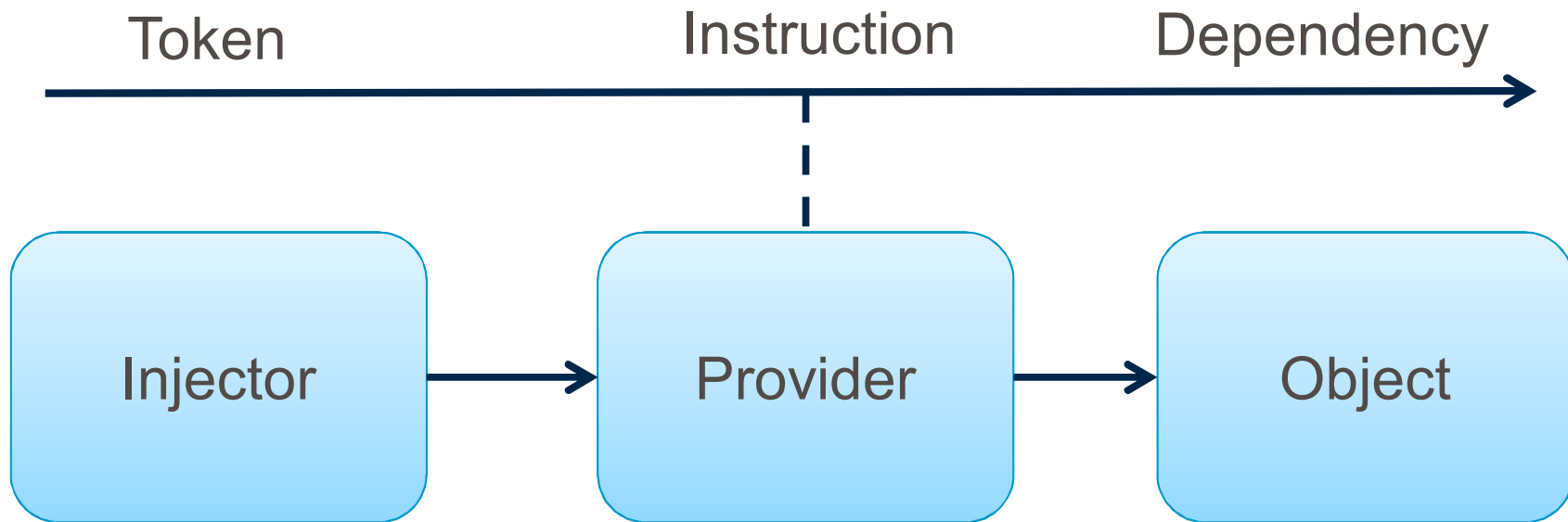
- ServiceDemo
- Registering-and-Injecting-Service



Providers

- Providers are usually singleton (one instance) objects, that other objects have access to through dependency injection (DI).
- A provider describes what the injector should instantiate a given token, so it describes how an object for a certain token is created.
- Angular 2 offers the following type of providers:
 - A class provider generates/provides an instance of the class (*useClass*).
 - A factory provider generates/provides whatever returns when you run a specified function (*useFactory*).
 - Aliased Class Provider (*useExisting*)
 - A value provider just returns a value (*useValue*).

Angular 2 DI System



Demo

- ClassProvider
- ValueProvider
- FactoryProvider



Observables

- Observables is like an array whose items arrived asynchronously.
- Observable help to manage asynchronous data, such as data coming from a backend service.
- Observables are proposed feature for ES 2016 the next version of JavaScript. To use observables now angular uses a third party library called reactive extensions.
- Observables are used with in angular itself including angular's event system and its http client service
- A method can be subscribed to an observable to receive asynchronous notifications as new data arrives.

Introducing RxJs

- RxJs stands for Reactive Extensions for Javascript, and its an implementation of Observables for Javascript.
- It is a ReactiveX library for JavaScript.
- It provides an API for asynchronous programming with observable streams.
- ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming.

```
var source = Rx.Observable.interval(1000).map(num=>['1','2','3','A','4','5','6'][num]);  
var result = source.map(x=>parseInt(x)).filter(x=> !isNaN(x));  
result.subscribe(x=>console.log(x));
```

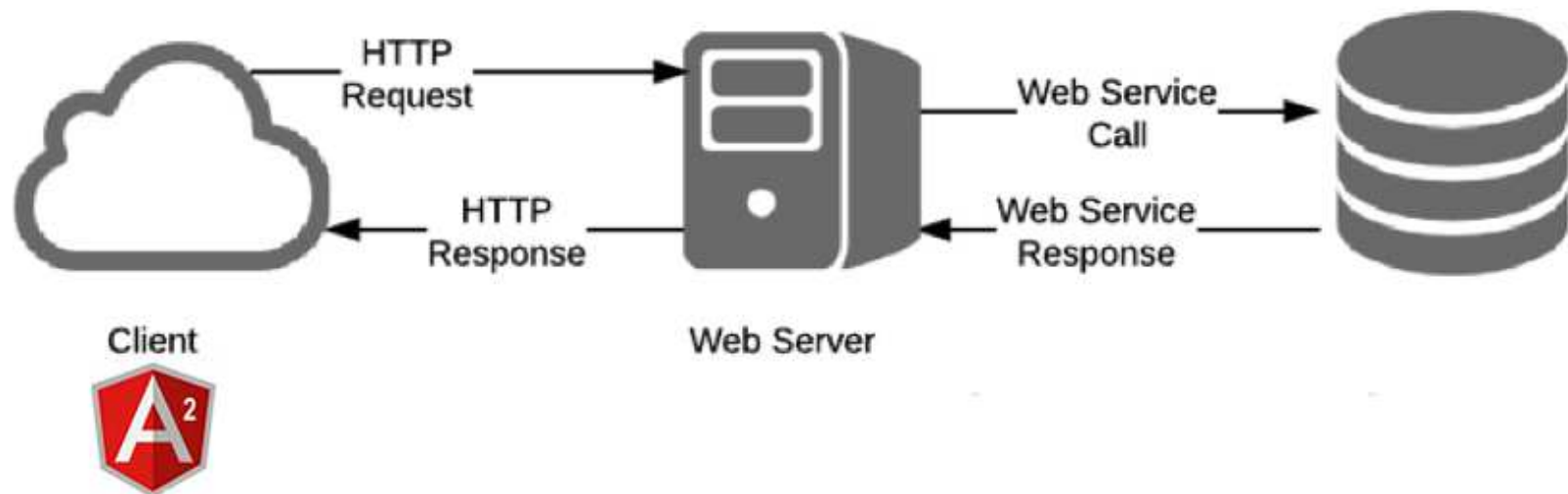
Demo

- Observables
- RenderingObservableWithpipe



Angular2 HTTP

- Angular applications often obtain data using http
- Application issues http get requests to a web server which returns http response to the application.
- Application then processes that data



Http Class

- Performs http requests using `XMLHttpRequest` as the default backend.
- Http is available as an injectable class.
- Calling request returns an Observable which will emit a single Response when a response is received.
- To work with Http Class
 - Include Angular 2 Http script (http.dev.js) in index.html
 - Include script tag for the reactive extensions (Rx.js) in index.html
 - Register HTTP_PROVIDERS
 - Import RxJS

Catch Operator

- Reacts to the error case of an Observable.
- Need to return a new Observable to continue with

```
export class UserProxy{
  constructor(private http :Http){}
  load(){
    return this.http
      .get('http://api.randomuser.me/10')
      .map(res =>res.json())
      .catch(this.logAndPassOn);
  }
  private logAndPassOn (error: Error) {
    console.error(error);
    return Observable.throw(error);
  }
}
```

Communication with JSONP

- Angular provides us with a JSONP services which has the same API surface as the Http.
- Only difference that it restricts us to use GET requests only.
- JSONP service requires the JSONP_PROVIDERS.

Server Simulation

- To enable our server simulation, we replace the XHRBackend service with the in-memory web api backend.
- The in-memory api must to implements ConnectionBackend

```
bootstrap(App,[ HTTP_PROVIDERS,  
  // in-memory web api providers  
  provide( XHRBackend, { useClass: InMemoryBackend } )  
]);
```

Demo

- HttpDemo



Summary

- Angular's dependency injection system creates and delivers dependent services "just-in-time".
- Angular ships with its own dependency injection framework.
- Angular creates an application-wide injector for us during the bootstrap process no need to create explicitly.
- A provider *provides* the concrete, runtime version of a dependency value. The injector relies on **providers** to create instances of the services that the injector injects into components and other services.
- We must register a service *provider* with the injector, or it won't know how to create the service.



Summary

- RxJS is a third party library, endorsed by Angular, that implements the asynchronous observable pattern.
- Observables are used with in angular itself including angular's event system and its http client service.
- The Angular HTTP library simplifies application programming with the XHR and JSONP APIs.
- The Angular Http client communicates with the server using a familiar HTTP request/response protocol.



Angular 2

Angular 2 Forms

Forms in Angular 2

- Forms are probably the most crucial aspect of your web application.
- An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors using the following tools
 - **FormControls**: Encapsulate the inputs in forms and give the objects to work with them
 - **Validators**: Gives the ability to validate inputs
 - **Observers**: watch form for changes and respond accordingly
- To use the Forms library we need to import *FormsModule* or using *ReactiveFormsModule* in *app.ts*.
 - FormsModule gives template driven directives such as ngModel and NgForm
 - ReactiveFormsModule gives directives like FormControl and FormGroup

Two ways to build forms in Angular 2

Template
Driven

Model
Driven

Template Driven Forms

- Forms build by writing templates in the Angular template syntax with the form-specific directives and techniques are called as Template Driven Forms.
- Form is setup and configured in HTML Code.
- Angular2 “infers” the FormGroup from HTML Code
- Form data is passed via `ngSubmit()`
- *FormsModule* gives the template driven directives such as:
 - `ngModel` : Bind the model to form control.
 - `NgForm` : Angular provide a way in which form is exported as `ngForm`. We then assign the form data to local variable preceded with `#`.

Demo

- TemplateDriven-Forms



Model Driven Forms

- Model-driven forms are creating a representation of form in code i.e. forms end up as properties on components
- Model-driven forms enable us to test our forms without being required to rely on end-to-end tests.
- It's important to understand that when building reactive/model-driven forms, Angular doesn't magically create the templates
 - Reactive forms are more like an addition to template-driven forms
- *ReactiveFormsModule* gives the model driven directives such as:
 - `formControl`
 - `ngFormGroup`

FormControl and FormGroup

- The two fundamental objects in Angular 2 forms are FormControl and FormGroup.
- FormControl
 - A FormControl represents a single input field - it is the smallest unit of an Angular form
 - FormControls encapsulate the field's value, and states such as if it is valid, dirty (changed), or has errors.
- FormGroup
 - FormGroup is used to manage multiple FormControls
 - FormGroup provides a wrapper interface around a collection of FormControls.

FormBuilder

- FormBuilder class helps to reduce repetition and clutter by handling details of control creation.
- Inject a FormBuilder into the constructor.
- FormBuilder.group is a factory method that creates a FormGroup.
 - FormBuilder.group takes an object with properties, which is nothing but FormControl
 - FormControl values can be retrieved using *value* property of FormBuilder.group instance

Demo

- ModelDriven-Forms
- ModelDriven-Forms-FormBuilder



Form Validation

- Form Validation is used to improve overall data quality by validating user input for accuracy and completeness.
- Form element carries the HTML validation attributes like required, minlength, maxlength and pattern. Angular interprets those and adds validator functions to the control model.
- Angular exposes information about the state of the controls including whether the user has "touched" the control or made changes and if the control values are valid.
- We can add multiple validators in a single field using `Validators.compose`
- To look up a specific validation failure use the `hasError` method

Demo

- FormValidation



Custom Validations

- Angular allows to create custom validators as well.
- A validator: - Takes a FormControl as its input and - Returns a StringMap<string, boolean> where the key is “error code” and the value is true if it fails

```
function pinCodeValidator(control: FormControl): { [s: string]: boolean } {  
    if (!control.value.match(/^\d{6}$/)) {  
        return {invalidPinCode: true};  
    }  
}
```

Demo

- CustomValidation



Summary

- An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.
- An Angular form has two parts: an HTML-based template and a component class to handle data and user interactions programmatically.
- We can build almost any form using an Angular template itself
- The NgModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state.
- We can add multiple validators in a single field using Validators.compose
- To look up a specific validation failure use the hasError method



Angular 2

Internals of Angular 2

Change Detection

- Angular 2 can detect when component data changes and then automatically re-render the view to reflect that change through its change detection implementation.
- Angular 2 at startup time overrides several low-level browser APIs for instance *addEventListener* to run change detection and update the UI.
- This low-level patching of browser APIs is done by a library shipped with Angular called **Zone.js**
- The following frequently used browser mechanisms which usually cause changes are patched to support change detection
 - All browser events (click, mouseover, keyup, etc.)
 - Timers: `setTimeout()` and `setInterval()`
 - Ajax requests : XHR - Fetching data from a remote server
- Zone.js intercepts all ASYNC operations
- Angular has its own zone called NgZone to control Change Detections

How Change Detection works

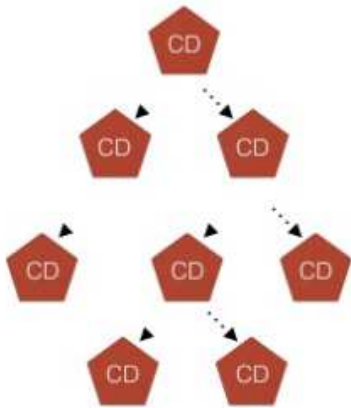
- Angular application will have a number of components that will interact with each other which creates a dependency tree.
- Each component gets a change detector, so we end up with a tree of change detectors.
- When one of the components change, no matter where in the tree it is, a change detection pass is triggered for the whole tree.
 - This happens because Angular scans for changes from the top component node, all the way to the bottom leaves of the tree (Unidirectional flow)
 - It gives a impression that this check may be a very expensive operation but angular generates VM friendly code for better performance which can perform hundreds of thousands of such checks in a few milliseconds.

Customizing Change Detection

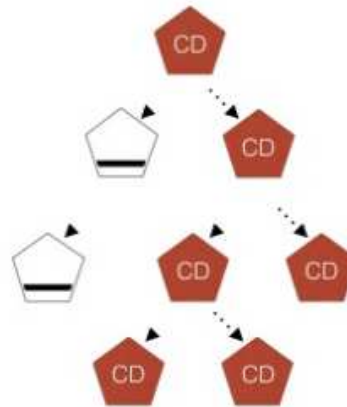
- There are times that the built-in or default change detection mechanism may be overkill.
- Angular provides mechanisms for configuring the change detection system such to get very fast performance.
 - Change detection should be optimized using Immutable Data and Observables
 - If immutable objects or observables used it checks only the parts of tree.
- Change detector behavior can be changed by telling a component that it only should be checked if one of its input values change by setting its *changeDetection* attribute to *ChangeDetectionStrategy.OnPush*
 - The default value for *changeDetection* is *ChangeDetectionStrategy.Default*.

Angular 2 change detection system

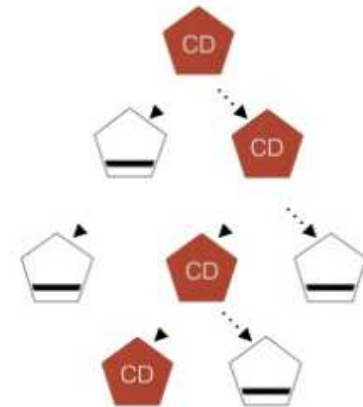
Default Change detection



Change detection with
Immutable Objects



Change detection with
Observable objects



Demo

- ChangeDetection



Summary

- Angular 2 uses Change Detection, In order to make the view react to changes on components state.
- Angular 2 can detect when component data changes and then automatically re-render the view to reflect that change though its change detection implementation.
- The default value for changeDetection in Angular is ChangeDetectionStrategy.OnDefault
- Angular also provides mechanisms for configuring the change detection system such to get very fast performance.



Angular 2

Routing

Routing

- Routing means loading sub-templates depending upon the URL of the page.
- We can break out the view into a layout and template views and only show the view which we want to show based upon the URL the user is accessing.
- Routes are a way for multiple views to be used within a single HTML page. This enables you page to look more "app-like" because users are not seeing page reloads happen within the browser.
- Defining routes in application can:
 - Separate different areas of the app
 - Maintain the state in the app
 - Protect areas of the app based on certain rules

AngularJS Routes

- AngularJS routes enable us to create different URLs for different content in our application.
- Having different URLs for different content enables the user to bookmark URLs to specific content.
- In Angular 2 routes are configured by mapping paths to the component that will handle them.
- For instance, let consider an application with 2 routes:
 - A main page route, using the `/#/home` path;
 - An about page, using the `/#/about` path;
 - And when the user visits the root path (`/#/`), it will redirect to the home path.

Routing Setup

- To implement Routing to Angular Application

- Import RouterModule and Routes from '@angular/router'

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

- Define routes for application

```
const routes: Routes = [ { path: 'home', component: HomeComponent }];
```

- Install the routes using RouterModule.forRoot(routes) in the imports of NgModule

```
imports: [ BrowserModule, RouterModule.forRoot(routes)]
```

Components of Angular 2 routing

- There are three main components are used to configure routing in Angular

Routes

- Describes the routes application supports

RouterOutlet

- A “placeholder” component that gets expanded to each route’s content

RouterLink

- Directive is used to link to routes

Routes

- To define routes for application, create a Routes configuration and then use RouterModule.forRoot(routes) to provide application with the dependencies necessary to use the router.
 - path specifies the URL this route will handle
 - component maps to the Component and its template
 - optional redirectTo is used to redirect a given path to an existing route

```
const routes: Routes = [  
  { path: '', redirectTo: 'home', pathMatch: 'full' },  
  { path: 'home', component: HomeComponent },  
  { path: 'about', component: AboutComponent },  
  { path: 'contact', component: ContactComponent },  
  { path: 'contactus', redirectTo: 'contact' },  
];
```

RouterOutlet

- The router-outlet element indicates where the contents of each route component will be rendered.
- RouterOutlet directive is used to describe to Angular where in our page we want to render the contents for each route

```
@Component({  
  selector: 'my-app',  
  template: `<div class="container">  
    <router-outlet></router-outlet>  
  </div>`  
})
```

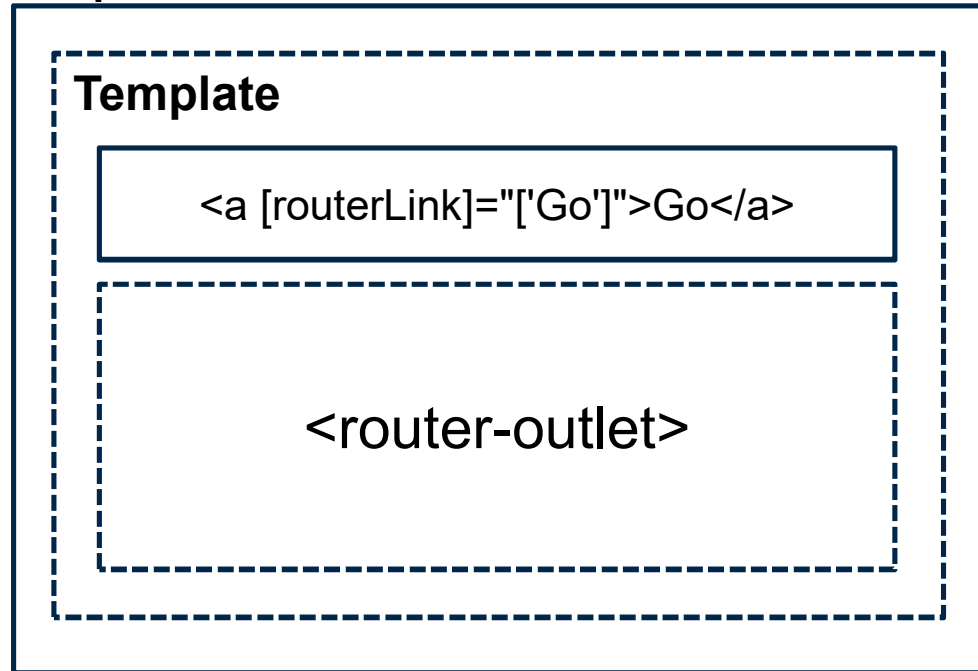
RouterLink

- It generates link based on the route path.
- routerLink navigates to a route

```
<div>  
  <a [routerLink]="['Home']">Home</a>  
  <a [routerLink]="['About']">About Us</a>  
</div>
```

RouterOutlet & RouterLink

Component :



HTML : `<a [routerLink]='['Go']">Go`

Code : `router.navigate(['Go']);`

Routing Strategies

- The way the Angular application parses and creates paths from and to route definitions is now location strategy.
- HashLocationStrategy ('#/')
- PathLocationStrategy (HTML 5 Mode Default)

```
//import LocationStrategy and HashLocationStrategy
import {LocationStrategy, HashLocationStrategy} from '@angular/common';

//add that location strategy to the providers of NgModule
providers: [
  { provide: LocationStrategy, useClass: HashLocationStrategy }
]
```

Route Parameters

- Route Parameters helps to navigate to a specific resource. For instance product with id 3
 - /products/3
- route takes a parameter by putting a colon : in front of the path segment
 - /route/:param
- To add a parameter to router configuration and to access the value refer the code given below

```
const routes: Routes =([
  { path:'/products/:id', name:'Product', component:ProductComponent }
])

/*To access the parameter value */
routeParams.get('id')
```

ActivatedRoute

- In order to access route parameter value in Components, we need to import ActivatedRoute

```
import { ActivatedRoute } from '@angular/router'
```

- inject the ActivatedRoute into the constructor of our component

```
export class ProductComponent {  
  id: string;  
  
  constructor(private route: ActivatedRoute) {  
    route.params.subscribe(params => { this.id = params['id']; });  
  }  
}
```

Demo

- RoutingDemo



Summary

- Routing means splitting the application into different areas usually based on the rules derived from the current URL in the browser
- Defining routes in the application helps us to:
 - Separate different areas of the application
 - Maintains the state in the application
 - Protect the areas of the application based on certain rules
 - Bookmarking a page
- RouterOutlet is a placeholder component that gets expanded to each route's content
- The three main components that we use to configure routing in Angular are : Routes, RouterOutlet and RouterLink
- RouterLink directive is used to link to routes



Summary

- Angular2 by default support HTML5 mode routing
- PathLocationStrategy and HashLocationStrategy are the two concrete strategy class used in Angular application which parses and create paths from and to route definitions.
- Route Parametes helps to navigate to a specific resource.
- route takes a parameter by putting a colon : in front of the path segment
 - /route/:param
- In order to access route parameter value in Components, we need to import ActivatedRoute

