AngularJS 4

Compiled by Lakshman M N
Tech Consultant and Mentor
mnl@lakshman.biz

What is Angular?

- “Client-side framework written entirely in JavaScript”
- Open-source web application framework maintained by Google and community.
- Framework typically used to build single-page applications.

What is Angular?

- It is a “structural framework for dynamic web apps”
- Works with long established technologies/standards HTML, CSS and JavaScript.
- Helps build interactive, modern web applications by increasing abstraction between the developer and common web app development tasks.

History!

- AngularJS was originally developed in 2009 by Miško Hevery and Adam Abrons
- It was envisaged as the software behind an online JSON storage service, that would have been priced by the megabyte, for easy-to-make applications for the enterprise.
- The two decided to abandon the business idea and release Angular as an open-source library.

Why Angular?

- Many JavaScript frameworks compel us to extend from custom JavaScript objects and manipulate DOM
  - The developer requires knowledge of the complete DOM & force complex logic into JavaScript code.
- AngularJS augments HTML to provide native MVC capabilities.
  - The developer can encapsulate a portion of the page as one application instead of compelling the entire page to be an AngularJS application.

Angular! Why?

- Angular makes HTML more expressive
  - Powers HTML with features like if conditions, for loops and variables.
- Angular has powerful data binding.
  - Fields from the data model can be displayed, changes can be tracked and updates from the user can be processed.
- Angular promotes modularity by design.
  - Application becomes a set of building blocks making it easy to create and reuse content.
- Angular provides built-in support to integrate with remote back-end services.
  - Facilities include the ability to get and post data or execute server-side logic.

Angular?

- Angular is built for speed.
  - It has faster initial loads, faster change detection and improved rendering times.
- Angular is modern.
  - It leverages features provided in the latest JavaScript standards such as modules etc.
  - Helps build reusable UI widgets employing web component technologies.
  - It supports greenfield and legacy browsers (including IE9!)

The JavaScript Language

- The JavaScript language specification is officially called ECMAScript (ES)
- ES 3 is supported by older browsers.
- ES 5 is supported by most modern browsers.
- ES 6 specification approved recently has been renamed ES 2015.
  - Most browsers currently lack full support for ES 2015 and hence ES 2015 code must be “**transpiled**” to ES5.

Angular language choices

- Any of the compiled JavaScript languages can be used to author Angular applications.
- **ES 5**
  - Version of JavaScript is the most common choice.
    - Code runs in the browser without transpilation.
- **ES 2015**
  - can be used to employ new features such as classes, let statement etc. and then transpiled to ES 5 before running.
- **TypeScript**
  - Superset of JavaScript and must be transpiled
  - TypeScript provides strong typing (everything has a data-type)
  - TypeScript has good IDE support that provides syntax checking, code navigation etc.
  - Angular has been authored using TypeScript.

ES 2015

- ES5 has been around since 2009.
  - It is supported by most of the popular browsers.
- In June 2015 a new specification of the JavaScript standard was approved that contains a lot of new features.
- It is called ECMAScript 2015 or also called ES6 as it is the 6th edition of the standard.
  - ECMAScript is the official name of the JavaScript language.
- Existing browsers don't support most of the features of ES6 yet.

ES 2015…

- Feature support across browsers varies widely.
- Are we expected to wait a few years and commence using ES6 after browsers start offering support?
  - http://kangax.github.io/compat-table/es6/
- Fortunately not!
- There are tools that can convert ES6 code into ES5 code.
- We write code using the new useful features of ES6 and generate ES5 code that will work in most of the current browsers.

ES6 / ES 2015

ES6 brings a lot of new features, some of which include:

- Classes
- Arrow Functions
- Template Strings
- Inheritance
- Constants and Block Scoped Variables
- Modules

Classes

- Classes are a new feature in ES6, used to describe the blueprint of an object
  - They perceive the transformation of ECMAScript's prototypal inheritance model to a more traditional class-based language.
- ES6 classes offer a much nicer, cleaner and clearer syntax to create objects and deal with inheritance.
- The class syntax is not introducing a new object-oriented inheritance model to JavaScript.

Classes...

```
class Shape {
  constructor(type){
     this.type = type;
  }
  getType(){
      return this.type;
  }
}
```

- Use the **`class`** keyword to declare a class.
- **`constructor`** is a special method for creating and initializing an object.
  - There can only be one special method with the name **`constructor`**

ES6-Demo01.htm

Classes...

- The **`static`** keyword defines a static method for a class.
- Static methods are called without instantiating their class and are not callable when the class is instantiated.
- Static methods are often used to create utility functions for an application.

```
class Shape {
  constructor(type){
     this.type = type;
  }
  static getClassName(){
    const name = 'Shape';
   return name;
  }
}

console.log(Shape.getClassName());
```

ES6-Demo02.htm

Subclassing

- The **`extends`** keyword is used in class declarations to create a class as a child of another class.
- The **`super`** keyword is used to call functions on an object's parent.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Dog extends Animal {
  speak() {
    super.speak();
    console.log(this.name + ' barks.');
  }
  }

var d = new Dog('Mitzie');
d.speak();
```

ES6-Demo03.htm

`this` revisited

- In JavaScript '**`this`**' keyword is used to refer to the instance of the class.

```
var shape = {
  name: 'square',
  say: function(){
       console.log('This is say(): ' + this.name);

       setTimeout(function(){
              console.log('Inside setTimeout(): '
                     + this.name);
       }, 2000);
  }
};

shape.say();
```

- The **`this.name`** is empty when accessed within **`setTimeout`**.

ES6-Demo04.htm
ES6-Demo05.htm

Arrow functions

- ES6 offers new feature for dealing with **`this`**, "**arrow functions**" **=>**
  - Also known as *fat arrow*
- Some of the motivators for a *fat arrow* are:
  - One does not need to specify **`function`**
  - It lexically captures the meaning of **`this`**
- The fat arrow notation can be used to define anonymous functions in a simpler way.
- Helps provide the context to **`this`**

Arrow functions

```
var shape = {
      name: 'square',
      say: function(){
            console.log('This is say(): ' + this.name);

            setTimeout(() => {
                  console.log('Inside setTimeout(): '
+ this.name);
            }, 2000);
      }
};

shape.say();
```

ES6-Demoo6.htm

Arrow functions...

- Arrow functions do not set a local copy of **`this`**, **`arguments`** etc.
- When **`this`** is used in an arrow function, JavaScript uses the **`this`** from the outer scope.
- If **`this`** *should* be the calling context, do not use the arrow function.

let

- **var** variables in JavaScript are *function scoped*.
- This is different from many other languages(Java, C#) where variables are *block scoped*.
- In ES5 JavaScript and earlier, **var** variables are scoped to the function and they can "*see*" outside their functions into the outer context.

```
var foo = 123;
if(true){
    var foo = 456;
}
console.log(foo);  //456
```

ES6-Demo07.htm

let…

- ES6 introduces the **`let`** keyword to allow defining variables with true *block scope*.
- Use of the **`let`** instead of **`var`** gives a true unique element disconnected from what is defined outside the scope.

```
let foo = 123;
if(true){
 let foo = 456;
}
console.log(foo);  //123
```

ES6-Demoo8.htm

let…

- Functions create a new variable scope in JavaScript as expected.

```
var num = 123;
function numbers(){
    var num = 456;
}

numbers();
console.log(num);   //123
```

ES6-Demo09.htm

let…

- Usage of **let** helps reduce errors in loops.
- **let** is extremely useful to have for the vast majority of the code.
- It helps decrease the chance of a programming oversight.

```
var index = 0;
var myArray = [1,2,3];
for(let index = 0; index < myArray.length;
                index++){
console.log(myArray[index]);
}
console.log(index);   //0
```

ES6-Demo10.htm

const

- **const** is a welcome addition in ES6.
- It allows immutable variables.
- To use **const**, replace **var** with **const**

```
const num = 123;
```

- **const** is a good practice for both readability and maintainability.
- **const** declarations must be initialized

```
const foo;   //ERROR
```

const

- A **const** is block scoped like the **let**
- A **const** works with object literals as well.

```
const foo = { bar : 123 };
foo = { bar : 456 };  //ERROR
```

- **const** allows sub properties of objects to be mutated

```
const foo = { bar : 123 };
foo.bar = 456;  //allowed
console.log(foo);  // { bar : 456 }
```

Template Strings

- In traditional JavaScript, text that is enclosed within matching “ or ‘ marks is considered a string.
- Text within double or single quotes can only be on one line.
- There was no way to insert data into these strings.
- If there was a need it would have required concatenation that looked complex and not so elegant.
- ES6 introduces a new type of string literal that is marked with back ticks (`)

Template Strings

- The motivators for Template strings include
  - Multiline Strings
  - String Interpolation
- Multiline Strings

```
var desc = 'Do not give up \
\n Do not bow down';
```

  - with Template Strings

```
var desc = `Do not give up
Do not bow down`;
```

Template Strings...

- String Interpolation

```
var lines = 'Do not give up';
var html = '<div>' + lines + '</div>';
```

  - with Template Strings

```
var lines = 'Do not give up';
var html = `<div>${lines}</div>`;
```

- Any placeholder inside the interpolation **${ }** is treated as a JavaScript expression and evaluated.

ES6-Demo12.htm

Modules

- JavaScript has always had problem with namespaces.
  - Variables and functions can end up in the global namespace if not cautious.
- JavaScript lacks built-in features specific to code organization.
- Modules help resolve these issues.
- Angular 1 provides modules to help organize code and resolve some namespacing issues.

Modules in ES6

- In ES6 each module is defined in its own file.
- The functions and variables defined in a module are not visible outside unless explicitly exported.
- A module can be coded in a way such that only those values that need to be accessed by other parts of the application need be exported.
- Modules in ES6 are declarative in nature.
- To export variables from a module use **`export`**.
- To consume the exported variables in a different module use **`import`**.

Working with Modules

```
function getRandom(){
      return Math.random();
}

function add(n1, n2){
      return(n1+n2);
}

export {getRandom, add}
```

Utils.js

- The **`export`** keyword is used to export the two functions.
- The exported functions can be renamed while exporting

```
export {getRandom as random, add as sum}
```

Utils.js

Working with Modules...

```
import { getRandom, add } from 'Utils';

 console.log(getRandom());
 console.log(add(1,2));
```

useUtils.js

- This imports the exported values from the module 'Utils'.
- A single value can be imported as well

```
import { add } from 'utils';
```

TypeScript

- An open source language.
- Superset of JavaScript.
- Compiles to plain JavaScript through transpilation.
- Implements ES 2015 class based object orientation.
- Strongly typed, therefore every function, variable, and parameter can have a data-type.
  - Uses type definition files to determine appropriate types when using JavaScript libraries that are not strongly typed.

A typescript file

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

- The typescript source file sports a **`.ts`** extension.
- The typescript compiler **`tsc`** can be used to compile a typescript source file into ES5.
- The resulting **`.js`** file resembles the following

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

Demo01.ts

Working with `tsc`

- **`tsc`** can handle multiple files as arguments.

  `tsc Demo01.ts Demo02.ts`

- This results in two corresponding **`.js`** files.
- Typescript has a means to tell **`tsc`** what to compile and other settings using **`tsconfig.json`** file.
- When **`tsc`** is run, it looks for **`tsconfig.json`** file and uses the rules to compile.

Demo02.ts

TypeScript features

- **Types**
- Though JavaScript does provide types, they're “duck typed”.
  - The programmer does not need to think about them.
- JavaScript’s types also exist in TypeScript
  - **`boolean`**
  - **`number`**, **`NaN`**
  - **`string`**
  - **`[]`** Arrays
  - **`{}`** Object literal
  - **`undefined`**
- TypeScript also adds
  - **`enum`** enumerations like **`{One, Two, Three}`**
  - **`any`** use any type
  - **`void`** nothing

Primitive Types

```
let isComplete: boolean = false;
let width: number = 6;
let name: string = 'Doe';
let list: number[] = [1,2,3];
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
let unCertain: any = 4;
```

Functions

```
function getDayName(dayNumber: number):
                        string {

    const daysArray: string[] = ['Sunday',
            'Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday'];

    const dayName: string = daysArray[new
            Date().getDay()];

    return dayName;
}
```

Demo03.ts

Function parameters

- JavaScript functions can routinely accept optional parameters.
- TypeScript provides support for the same albeit slightly differently.
- Using **?** tells **tsc** that the corresponding parameter is an optional one.

```
function logData(data: string, isVerbose?: boolean){
    if(isVerbose){
        console.log("Verbose data " + data);
    }
    else{
        console.log(data);
    }
}

logData("Data logging");
logData("Data logging", true);
```

Demo04.ts

Angular - Characteristics

- Support for ES6
- Code is more readable and re-usable with components.
- Performance enhancements
- Better mobile support.
- Controllers and scope are no more.
- Properties are now bound to components instead of **`$scope`**.
  - Everything is in a component as opposed to passing **`$scope`** between views.

Angular - Characteristics

- Data flow
  - One-way data binding from a component to the UI
  - One-way data binding from the UI to component using events.
  - Two-way data binding still available but infrequently used.
    - It internally employs the two separate one-way bindings.
- Templates
  - Very similar to Angular 1
  - Syntactical changes to make things clear
  - New built-in directives

Architecture Overview

- Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript.
- The framework is comprised of many libraries, core and optional.
- Angular applications are authored by
  - composing HTML templates,
  - component classes to manage those templates
  - Adding application logic in services
  - Grouping services and components in modules.

Setting up the environment

- Setting the development environment for Angular requires a few steps.
- The first step requires the installation of NodeJS.
  - Installer is downloadable from https://www.nodejs.org/
- **NodeJS** is a open-source, cross platform JavaScript runtime environment.
- ***npm*** (Node Package Manager)
  - A command line utility that interacts with a repository of open source projects.
  - Allows for installation of libraries, packages and applications along with dependencies.

Setting up an Angular application

- Create an application folder
- Create the **`tsconfig.json`** file to configure the typescript compiler for generating JavaScript from the project's files.
- Create the **`package.json`** file to identify package dependencies for the project.
- Create the **`typings.json`** file to provide additional definition files for libraries that the TypeScript compiler does not recognize.
- Install the libraries and typings file
- Create the host Web page (index.html)
- Create the **`main.ts`** file to bootstrap the angular application with the root component.

Setting up an Angular app…

- Options include:
  - Manually perform each step
    - Detailed instructions for each step available in the angular documentation at www.angular.io
  - Download the results of these steps
    - Quickstart files from www.angular.io
    - https://github.com/angular/quickstart
  - Use tooling such as AngularCLI
    - A command line tool for generating the setup files and boilerplate code for an Angular application.
    - https://github.com/angular/angular-cli

Project setup with Angular CLI

- The Angular CLI is a tool to initialize, develop, scaffold and maintain Angular applications.
- Angular CLI makes it a breeze to create an application that works using all the best practices.
- Installation of Angular CLI

```
npm install –g @angular/cli
```

- Generating an Angular project

```
ng new <Project-Name>
```

Project setup with Angular CLI

- Serving an Angular project via a development server

  `cd <Project-Name>`

  `ng serve`

  - The app is served at **`http://localhost:4200`**
  - The app will automatically reload if any of the source files are modified.
  - The http `host` and `port` used by the development server can be altered by

  `ng serve --host 0.0.0.0 --port 1122`

Angular CLI dependencies

- core-js
  - Modular standard library for JavaScript
  - Includes polyfills for ECMAScript 5
    - ES 6: promises, collections, iterators
  - Only needed features can be required without polluting the global namespace.
- zone.js
  - Angular does not have a **`$digest`** and hence zone.js helps inform Angular of changes to data.

Angular CLI dependencies

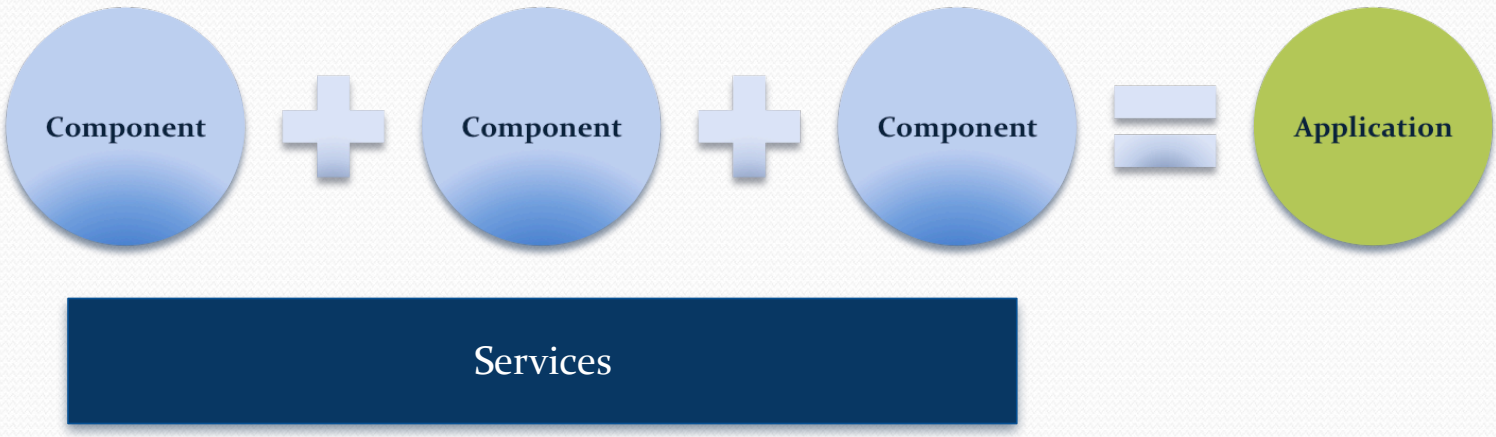
- webpack
  - webpack is a module bundler for JavaScript applications.
  - It helps package all the modules needed by the application into a small bundle, to be loaded by the browser.
- webpack-dev-server
  - Lightweight development only node server that provides live reloading
  - The server makes it easy to serve SPAs.

Architecture Overview

- The application is launched *by bootstrapping the root module*.
- An Angular application can comprise of several main building blocks
  - Modules
  - Components
  - Templates
  - Metadata
  - Data Binding
  - Directives
  - Services

Anatomy of an Angular Application

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



The file structure

- A bare-bones Angular application comprises of the following files:
- **`app/app.component.ts`**
  - This is where we define our root component
- **`app/app.module.ts`**
  - The entry angular module to be bootstrapped
- **`index.html`**
  - The page the component will be rendered in
- **`app/main.ts`**
  - The glue that combines the component and the page together.

Module

- Angular apps are modular and Angular has its modularity system called modules
- Every Angular app has at least one module, the root module conventionally named **`AppModule`**.
  - Most applications have many feature modules.
- An Angular module is a class with an **`@NgModule`** decorator.

Decorator

- A decorator is a function that adds metadata to a class, its members or its method arguments.
- The scope of the decorator is limited to the feature it decorates.
- A decorator is always prefixed with an **@**
- Angular has several built-in decorators.

`@NgModule()`

- The decorator is applied by positioning it immediately prior to the feature being decorated. (eg a class)

NgModule

- **NgModule** is a decorator function that accepts a single metadata object.
- The properties of this metadata object describe the module.
- Important properties include
  - **declarations** – indicate the view classes that belong to this module.
    - Angular has three kinds of view classes: **components**, **directives** and **pipes**.

NgModule

- **exports** – the subset of declarations to be accessible in the component templates of other modules.
- **imports** – other modules whose exported classes are required by component templates in the current module
- **providers** – used to create services that this module contributes to the global collection of services.
- **bootstrap** – the main application view, called the root component, that hosts all other app views.

A root module (app.module.ts)

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/
platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Bootstrapping

- An application in Angular is launched by bootstrapping its root module.
- Commonly the **`AppModule`** is bootstrapped in a **`main.ts`** file.

```
import { platformBrowserDynamic } from '@angular/
  platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

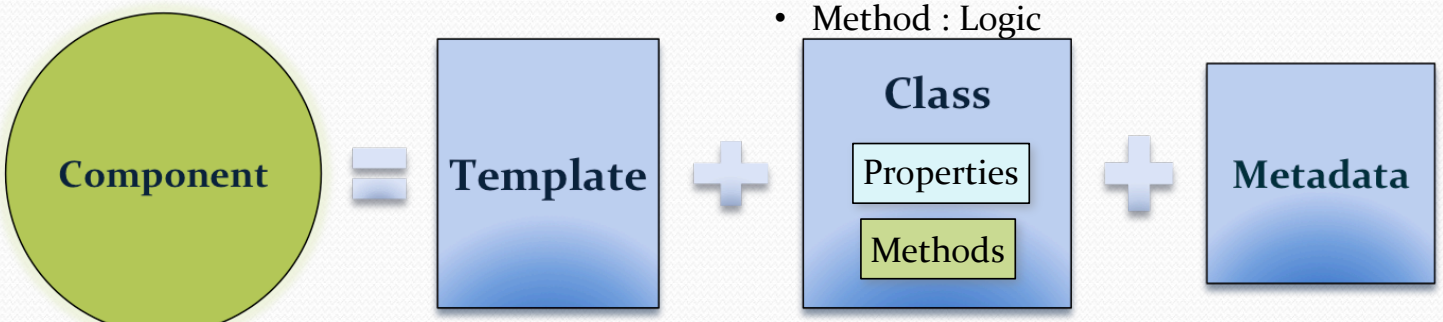
Angular vs JavaScript Modules

- An Angular module is a class decorated with **`@NgModule`**
  - This is a fundamental feature in Angular
- JavaScript has a module system for managing collections of JavaScript objects.
- In JavaScript each file is a module and all objects defined in the file belong to that module.
- The module declares objects to be public by marking them with the **`export`** keyword
- Other JavaScript modules use **`import`** statements to access public objects from other modules.

Angular libraries

- Angular ships as a collection of JavaScript modules that can be considered as library modules.
- A library module ***can*** comprise of
  - Components
  - Directives
  - Services
  - Data
  - Functions
- An Angular library name begins with the **`@angular`** prefix

An Angular Component



- Code supporting the view
- Created with TypeScript
- Properties : Data
- Method : Logic

- View layout
- Created with HTML
- Includes Bindings and Directives

- Extra data for Angular
- Defines a class as a Angular Component
- Defined with a decorator

Creating a Component

- Import the Component function (decorator) from **`@angular/core`**.

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

- Define the component metadata using the **`@Component`** decorator.

```
@Component({
    selector: 'random-quote',
    template: `
    <div><h2>Random Quote</h2>
    <p>{{quote.line}} - {{quote.author}}</p></div>`
})
```

Creating a Component...

- The **`@Component`** decorator is followed by the component class that is exported.

```
export class RandomQuoteComponent{
    quotes: Object[] = [...]
    constructor(){
    }
}
```

- All this content is saved in a file **`<name>.component.ts`**

Randomquote.component.ts

Using the component

- The component created can be used as a part of the **AppComponent**'s template.

```
@Component({
    selector: 'my-app',
    template: `<h1>My First Angular App</h1>
     <random-quote></random-quote>`
})
export class AppComponent { }
```

app.component.ts

Using the component...

- The component needs to be imported into the **AppModule**.

```
import { RandomQuoteComponent } from './
  randomquote.component';
```

- The component needs to be included in the declarations section of the metadata in the **@NgModule** decorator.
  - This is needed to tell Angular to include the corresponding class associated with the html element defined in the component's selector.

```
declarations: [AppComponent , RandomQuoteComponent]
```

app.module.ts

Styling a Component

- A component typically needs its own styles in order for it to be termed self-contained.
  - The styles should be packaged and scoped to the component.
- A component can be associated with CSS styles in three ways:
  - Component inline styles
  - Style urls
  - Template inline styles

Component inline styles

- Styles can be added to a component by adding a styles property to the **@Component** decorator.

```
@Component({
    selector: 'random-quote',
    styles: [`.quote {
        background-color: #ffff00;
    }
    `],
    template: `
    <div><h2>Random Quote</h2>
    <p class="quote">{{quote.line}} - {{quote.author}}
  </p></div>`
})
```

randomquote1.component.ts

Component inline styles...

- In the browser Angular takes the defined styles and writes them into the head of the HTML document.

```
<head>
  <style>
  .quote {
        background: #ffff00;
    }
  </style>
</head>
```

Style urls

- In an ideal world styles are not entwined with our application code.
- The <link> tag allows us to fetch a stylesheet from a server.
- Angular allows us to define **`styleUrls`**, so that styles don't have to be written into the component.
- Styles defined in a **`.css`** file can be fetched and employed by the component.

Style urls

- The **.css** file resides in the same folder as the **index.htm** in the context of our example.

```
@Component({
    selector: 'random-quote',
    styleUrls: ['./
randomquote.component.styles.css'],
    template: `
    <div><h2>Random Quote</h2>
    <p class="quote">{{quote.line}} -
{{quote.author}}</p></div>`
})
```

randomquote2.component.ts

Style urls

- In the case of style urls, Angular fetches the style resources, takes the text response, inlines and and writes them into the head of the HTML document.

```
<style>
  .quote {
    background-color: #ffff00;
    text-transform: capitalize;
  }
</style>
```

Template inline styles

- Styles can be written directly into the DOM.
- It is quite common to include styles directly into the template of a component.

```
@Component({
    selector: 'random-quote',
    template: `
    <style>.quote
    {
        background-color: #ffff00;
        text-transform: capitalize;
    }
    </style>
    <div><h2>Random Quote</h2>
    <p class="quote">{{quote.line}} - {{quote.author}}</p></
  div>`
})
```

randomquote3.component.ts

Template inline styles

- Template inline styles using the **<style>** tag in the template will be appended to the head of the HTML document.
- The style can also be written as normal inline styles in the template elements.

```
@Component({
    selector: 'random-quote',
    template: `
    <div><h2>Random Quote</h2>
    <p style="background-color:#ffff00;">{{quote.line}} -
  {{quote.author}}</p></div>`
})
```

Using Components (Composition)

- Creating a component to be used in another component.

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

@Component({
    selector: 'quote-header',
    template: `<h2>Random Quote</h2>`
})

export class QuoteHeader{
    constructor(){}
}
```

Components (Composition)

- Using the component in another component

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

@Component({
    selector: 'random-quote',
    template: `
    <div><quote-header></quote-header>
    <p>{{quote.line}} - {{quote.author}}</p></
  div>`
})

export class RandomQuoteComponent{
}
```

Components (Composition)...

- The component needs to be imported in **app.module**.
- It ***must*** be imported before the component that is using this component.

```
import { QuoteHeader } from './
  quoteheader.component';
import { RandomQuoteComponent } from './
  randomquote.component';
```

Data Binding

- Data binding is a mechanism for synchronizing what users see with application data values.
- A binding framework simplifies retrieving and populating values from HTML.
- Bindings can be declared between binding sources and target HTML elements.
- Angular provides multiple kinds of data binding.
- Bindings can be grouped into three categories depending on the direction of the data flow.

Data Binding categories

Data Direction	Syntax	Binding Type
One-way (from data source to view)	`{{ expression }}` `[target] = "expression"`	Interpolation Property Style
One-way (from view to data source)	`(target) = "statement"`	Event
Two-way	`[(target)] = "expression"`	Two-way

Bindings – Property Binding

- Expressions in Angular are employed to display properties of a component in the view.

```
@Component({
selector: 'my-comp',
template: `<h2> {{ pageTitle }} </h2>`
})
export class MyComponent {
pageTitle: string = 'Hello World!';
}
```

  - **{{ pageTitle }}** represents a property **pageTitle** in the component that uses interpolation.
- Behind the scenes Angular translates interpolations into property binding.
- Property binding helps bind properties of a DOM element to a property in the component.

Property Binding

- Angular offers a couple of syntax options for property binding.

```
<img [src]="imageUrl"/>
```

- Here we specify the DOM property in square brackets **[ ]**

```
<img bind-src="imageUrl"/>
```

- Here the DOM property is prefixed with **bind-**
- This is apart from the normal approach of using interpolation.

```
<img src="{{ imageUrl }}" />
```

- All the three syntactical usages are technically identical.

Property Binding

- The square bracket based usage appears simple and subtle.
- Interpolation based usage **{{ }}** works well when adding dynamic values with regard to content of DOM nodes.
  - It is extensively used when we need to render textual content / text between elements.
- Property binding is a **one-way** approach that is from the component to the DOM.
  - If the property in the component changes, Angular will update the DOM automatically.
  - Changes made in the DOM will not be reflected in the component.

quoteheader1.component.ts

Style Binding

- Style binding is a variation of property binding.

```
@Component({
    selector: 'quote-header',
    template: `<h2
  [style.backgroundColor]="isHeader ? 'yellow' :
  'aqua'">Random Quote</h2>
    <img
  [src]="imageUrl" [width]="imgWidth" [height]="i
  mgHeight"/>`
})
```

- Angular evaluates the binding expression and sets the background color value appropriately.

quoteheader2.component.ts

Event Binding

- DOM events are handled using parentheses **()** and template statements.
- When an event in parentheses is detected, the template statement is executed.

```
<button (click)="changeTitle()">Change Title</button>
```

- Alternatively, the canonical form can be used by prefixing the event name with **on-**

```
<button on-click="changeTitle()">Change Title</button>
```

- The target event **(click)** is in parentheses and the response to the event **changeTitle()** is to the right.

Two-way data binding

- Two-way data binding helps visualize updates to the view when model changes and vice-versa.
- This happens immediately which means that the model and view are always in sync.
- The view can be considered as an instant projection of the model.
- In Angular **`ngModel`** can be used to create two-way binding between a DOM property and a component property.

Two way data binding

- **`FormsModule`** is required to use **`ngModel`**.
- Import the **`FormsModule`** in the **`app.module.ts`** file

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

- Add the **`FormsModule`** into Angular module's **`imports`** list.

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

- To employ two way binding use **`[(ngModel)]`** in the context of a DOM node.

```
<input type='text' [(ngModel)]='pageTitle'
  name='pTitle'>
```

quoteheader5.component.ts

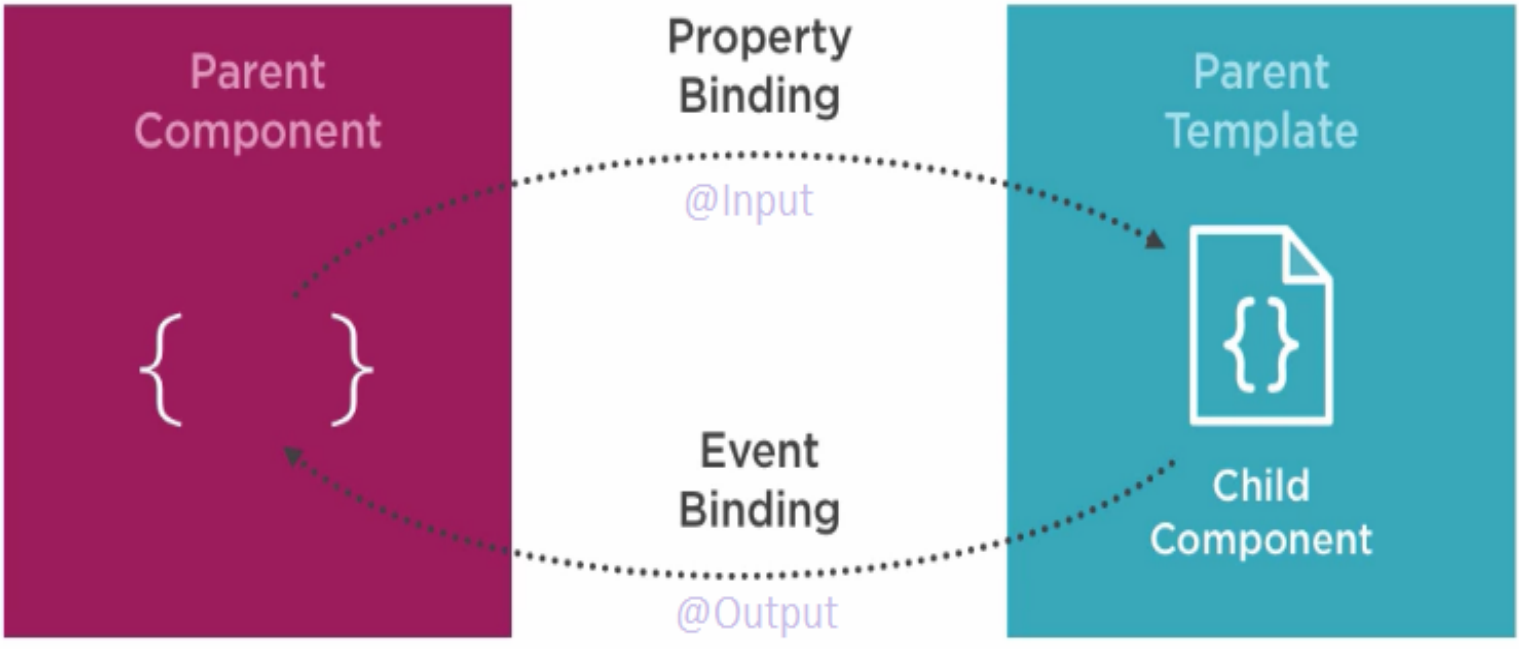
Component hierarchy

- Angular applications are generally constituted of a hierarchy of components.
- In Angular a component can receive data from its parent as long as the receiving component has indicated that it is willing to receive data.
- When a child component takes an action that a parent needs to know about, the child fires an event that is caught by the parent.

Input and Output

- **Input** specifies properties that can be set on a component.
  - Components allow **input** properties to flow in
- **Output** identifies the events that a component can fire to send information up the hierarchy.
  - **Output** events allow child components to communicate with a parent

Input and Output



Inputs and Outputs

```
1  import {Component, Input, Output, EventEmitter} from '@angular/core';
2
3  @Component({
4      selector: 'CounterComponent',
5      inputs: ['counter'],
6      outputs: ['counterChanged'],
7      template: `
8      <div (click)='counterEvent()'>
9      Click here!
10     </div>
11     `
12 })
13
14 export class CounterComponent{
15     counter: number;
16     public counterChanged = new EventEmitter();
17
18     constructor(){
19         console.log("Counter component");
20     }
21     counterEvent(evt:Event){
22         this.counterChanged.emit(this.counter);
23         this.counter++;
24     }
25 }
26
```

Inputs and Outputs...

- **`Input`**, **`Output`** and **`EventEmitter`** are imported for the component to be able to use them.
- The **`"inputs"`** property of the **`@Component`** decorator lists **`"counter"`** as a component property that can receive data.
- The **`"outputs"`** property lists **`"counterChanged"`** as a custom event that the component can **`emit,`** which the parent will be able to receive.
  - **`"counterChanged"`** is declared as and set to be an **`EventEmitter`**.
  - **`EventEmitter`** is a built-in class that ships with Angular and helps manage and fire custom events.
  - The custom event is fired when the **`<div>`** is clicked.

counter.component.ts

Using the component...

```
@Component({
    selector: 'quote-header',
    template: `Preview : {{ pageTitle }}
     <CounterComponent
  [counter] = "counterValue" (counterChanged)="handleEvent($event)">
  <CounterComponent>
`
})

export class QuoteHeader{
    pageTitle: string = "Random Quote";
    counterValue: number = 1;

    handleEvent(arg:Event){
        console.log("The counter value is " , arg);
    }
    constructor(){
    }
}
```

quoteheader6.component.ts

`@Input()` and `@Output()`

- Angular provides **`@Input`** and **`@Output`** decorators as an alternative to the inputs and outputs properties of the object in the **`@Component`** decorator.

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

@Component({
    selector: 'CounterComponent',
    template: `
    <div (click)='counterEvent()'>
    Click here!
    </div>`
})

export class CounterComponent{
    @Input() counter: number;
    @Output() counterChanged = new EventEmitter();

    counterEvent(evt:Event){
        this.counterChanged.emit(this.counter);
        this.counter++;
    }
}
```

counter1.component.ts

Summary

- Components in Angular represent a major new feature.
- Components are the key building block for Angular applications.
  - An application in Angular is a predominantly a collection of components.
- Components are defined using the **`@Component`** decorator.

Pipes

Pipes

- Pipes in Angular provide a mechanism to format the data to be displayed to the user.
- Angular provides a number of built-in pipes in addition to the ability to create new ones.
- Pipes are invoked in HTML using the | character inside the template binding {{ }}.
- Pipes were termed as filters in Angular 1.x

Using Pipes

- A pipe accepts data as input and transforms it into a desired output.
- **`uppercase`**
  - Converts the given string into uppercase

```
<span>{{location[0].country | uppercase }}</span>
```

- **`lowercase`**
  - Converts the given string into lowercase

```
<span>{{location[0].continent | lowercase }}</span>
```

- **`titlecase`**
  - Converts the first letter of each word in the string into uppercase

```
<span>{{someText | titlecase }}</span>
```

pipeDemo1app.component.ts

Currency pipe

- The currency pipe is used to format the numeric data as currency.
- It provides us the option of displaying a symbol or an identifier.
- The default currency option is that of the current locale.
- The usage of '**`true`**' indicates the usage of the currency symbol if available.

```
{{1234 | currency:'USD':true}}
```

pipeDemo2app.component.ts

Date pipe

- The date pipe allows to format the date in a required format style.
- It provides several built-in options.

```
{{ today | date:'medium'}}
```

- The date formatter enables customization of the date format.

```
{{ today | date:'EEEE, MMMM d, yyyy'}}
```

pipeDemo3app.component.ts

Array slice pipe

- The slice pipe in Angular only returns the items within a specific interval from a list.
- It can accept two parameters: the index from which to start and the index where to end.
  - The second parameter is optional.

```
Companies = [‘Apple’, ‘Bluestar’, ‘CapGemini’,
  ‘Dell’];
{{Companies | slice: “2”}}
{{Companies | slice: “2” : “3” }}
```

pipeDemo4app.component.ts

Numerical pipes

- There are a couple of pipes that help format numbers.
- **number**

```
myNumber = 176.775;
{{ myNumber | number : '3.1-2'}}
```

  - The pattern is {minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
- **percent**

```
{{ myNumber | percent }}
```

pipeDemo5app.component.ts

Custom pipes

- Angular allows the creation of custom pipes.

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

@Pipe({
    name: 'tempConvert'
})
export class TempConvertPipe implements PipeTransform{
    transform(value: number) : number {
        return  (value * 9/5) + 32;
    }
}
```

Custom pipes...

- A pipe is a class decorated with pipe metadata
- The pipe class implements the **`PipeTransform`** interface's **`transform`** method.
  - This method accepts an input value followed by optional parameters and returns the transformed value.
  - Angular is told that this is a pipe by applying the **`@Pipe`** decorator.
  - The **`@Pipe`** decorator allows to specify the pipe name used in template expressions.

pipeDemo6.component.ts
pipeDemo7.component.ts
pipeDemo6app.component.ts

Directives

Directives

- Angular templates are dynamic.
- Directives when rendered transform the DOM on the basis of instructions specified.
- A directive is a class with directive metadata.
  - The **`@Directive`** decorator is applied to attached metadata to the class.

Angular provides three kinds of directives:

- ***Components***
  - A component is in fact, a directive with a template.
  - Components are directives that are associated with the template directly.
  - **`@Component`** decorator extends the **`@Directive`**

Directives…

- ***Structural Directives***
  - These alter the DOM as it can add, remove or replace DOM elements.
  - **`NgIf`** – helps to show or hide an element.
- ***Attribute Directives***
  - Alter the element by changing its behaviour or the appearance.
  - **`NgClass`** – adds a class to an element

`@Component` VS `@Directive`

- A **`@Component`** requires a view whereas a **`@Directive`** does not.
- A component, rather than adding behaviour to an existing DOM element, creates its own view with behaviour.
- Component is used to break up the application into smaller components.
- Only one component can be present per DOM element.
- Multiple directives can be used per DOM element.
- A component is a directive with a template.

Custom directive

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

@Directive({
    selector: '[myFirstDirective]',
    host: {
        '(focus)': 'onFocus()',
        '(blur)': 'onBlur()'
    }
})
export class MyFirstDirective{
    constructor(private el: ElementRef, private renderer: Renderer){

    }
    onFocus(){
        this.renderer.setElementStyle(this.el.nativeElement, 'font-weight', 'bold');
        this.renderer.setElementStyle(this.el.nativeElement, 'width', '200px');
    }
    onBlur(){
        this.renderer.setElementStyle(this.el.nativeElement, 'font-weight', 'normal');
        this.renderer.setElementStyle(this.el.nativeElement, 'width', '120px');
    }
}
```

Demo1.directive.ts

Custom directive...

- The creation of a custom directive involves creating a class decorated with the **`@Directive`** decorator.
- The **`selector`** in the directive metadata is a CSS selector for the host element.
  - Usage of **`[ ]`** refers to an attribute.
- **`host`** is used to subscribe to events raised from the element on which the directive is applied.
  - Events and handlers are specified as key-value pairs.
  - Event names are in parentheses **`( )`** and the associated handlers are methods defined in the directive class.
- **`ElementRef`** provides access to the host element
- **`Renderer`** is used to modify the element.

Using a custom directive...

```
@Component({
    selector: 'my-app',
    template: `<h4>Directive in Angular</h4>
    <form>
    First Name : <input type="text"
name="txtFname" myFirstDirective />
    Last Name : <input type="text"
name="txtLname" myFirstDirective />
    </form>`
})
```

directiveDemo1app.component.ts

Built-in directives

- Angular provides a number of directives. Some of them include:
  - **`ngIf`**
  - **`ngSwitch`**
  - **`ngFor`**
  - **`ngClass`**

ngIf

- There are times when part of the view requires to be hidden or shown based on a condition.
- Consider a content author who has published courses online.
  - We may want to list the comments by viewers (in case there are any) below the course or mention that there are no comments yet.

```
<div *ngIf="comments.length>0">
        <p>Comments</p>
</div>
<div *ngIf="comments.length==0">
        <p>There are no comments yet</p>
</div>
```

ngIf

- The **ngIf** expression that evaluates to true will result in the corresponding node and its children inserted in the DOM.
- In case the expression evaluates to false the corresponding node and its children will be removed from the DOM.
- The **else** part ensures the specified node and its children are inserted into the DOM.

directiveDemo2app.component.ts

ngSwitch

- In a lot many programming languages there are two types of conditional statements, **`if..else`** and **`switch..case`**
- The same concept exists in Angular.
- **`ngSwitch`** is a built-in directive that is used to display an element and its children from a set of possible elements based on a condition.

directiveDemo4app.component.ts

ngFor

- **ngFor** is an Angular structural directive.
- **ngFor** is a repeater directive that can output a list of elements by iterating over an array.
  - The **ngFor** is similar to **ngRepeat** in Angular 1.x

```
<div *ngFor="let person of people”>
    <span>{{ person.firstname }}
        {{ person.lastname }}</span>
</div>
```

- “**let person of people**” creates a local variable **person**.
  - This variable can be used to reference the individual item in the collection.

directiveDemo5app.component.ts

The Leading Asterisk (*)

- Structural directives like **`ngIf`** use the HTML5 template element.
- The HTML5 **`<template>`** element is a mechanism for holding client-side content that is not to be rendered when a page is loaded.
- It may be instantiated during runtime using JavaScript.
- Based on the context Angular inserts the **`<template>`**'s contents into the DOM.
- The `*` makes it easier to read and write directives that modify HTML layout using templates.

The Asterisk (*)

- **ngFor**, **ngIf**, **ngSwitch** all add and remove element subtrees wrapped in template tags.

```
<p *ngIf="condition">
    Comments
</p>
```

*ngIf paragraph

```
<template [ngIf]="condition">
    <p>
        Comments
    </p>
</template>
```

ngIf paragraph with template

Elvis operator (?)

- Angular is a lot less forgiving about undefined references in binding expressions than Angular 1.x
- Angular introduces much stricter expression evaluation and generates errors “cannot read property ‘x’ of ‘undefined’” if one is not careful when creating template expressions.
  - This typically could be in cases where data is expected eventually but presently is undefined.
  - The Angular expression engine is unaware of this and throws an error.
- The Elvis operator allows us to specify bindings where the bound data will become available at a later time.

Elvis operator (?)

```
<span>
    Email : {{ person[0].work?.email }}
</span>
```

- The elvis operator in the above expression will wait/check if the data contains a property called '**email**' on '**work**'.

```
person = [
    {id: 1, firstname:'John', lastname: 'Doe'},
];
```

directiveDemo7app.component.ts

Transclusion in Angular

- *Transclusion* is an Angular 1.x term that represented taking content such as a text node or HTML and injecting it into a template at a specific entry point.
- This is done in Angular through a feature called ***Content Projection***.
- The word transclusion may be gone but the concepts remain.

Angular Content Projection

- **<ng-content></ng-content>** is like transclude in Angular 1.x
- Consider a round-border component whose template is

```
<div style='border:1px solid black;border-radius:4px'>
  <ng-content></ng-content>
</div>
```

- Use of the round-border component could resemble

```
<round-border>
  <h2>Hello World</h2>
</round-border>
```

- The 'Hello World' would then be embedded within the ** ** that has the rounded border on it.

rounded-border.component.ts

Form Validation

Background

- Forms help capture user input and facilitate interactivity in applications.
- Angular 1.x introduced some simplicity to deal with forms.
- Angular provides quite a few features with the aim of making the creation and validation of forms simple, intuitive and manageable.

Angular Forms

- Forms in Angular can be authored in two ways
- **Template Driven Forms**
  - Template driven forms are forms built entirely in the UI.
  - This is the traditional approach to build forms.
- **Model Driven Forms**
  - Similar to template driven forms but add an additional layer of functionality by having to declare the model of the form in the component class.
    - The additional functionality allows for greater control over the form.

Forms

- **View**

```
<input type="email" />
```

- **Model** – an instance of the `FormControl`
  - Has properties for validation such as
    - Value
    - Valid/invalid
    - Pristine/dirty
    - Touched/untouched
    - Errors
- **This is not the domain model**. This is a dedicated form model holding validation and form control state properties.

Enabling the Form API

- To access the Forms API **`FormsModule`** needs to be imported from `@angular/forms` and referenced in the module where it is to be used.

```
import { FormsModule }   from '@angular/forms';
…
@NgModule({
  imports: [ ..., FormsModule ],
  declarations: [ ... ],
  bootstrap: [ ... ]
})
export class AppModule {}
```

Template driven approach

```
<form>
  <div>
    Firstname: <input type="text" name="firstname" />
  </div>
  <div>
    Lastname: <input type="text" name=”lastname" />
  </div>
  <div>
    Age: <input type="text" name="Age" />
  </div>
</form>
```

Template driven approach

- In order to hook it Angular's Forms API a **`#form`** should be declared

```
<form #form="ngForm">
    ...
</form>
```

- To create bindings the **`ngModel`** attribute is added to the form controls.

```
<input type="text" ngModel name="firstname" />
```

- By adding the **`ngModel`** attribute, a **`FormControl`** instance is created behind the scenes by Angular.
- This is required to access the underlying DOM element value.

Grouping form controls

- Often it is necessary to group a number of form controls together.
- Controls can be bound as a form group.
- Form groups get the same properties as individual form controls (value, validity, touched/untouched etc)
  - If one child is invalid, the entire group is invalid.
- Form controls can be grouped by adding **`ngModelGroup`**

contact-formapp.component.ts
contact-form.component.ts
contact-form.component.html
contact-form-index.html

Showing Errors

- The "**#form**" provides us access to controls collection.
- A particular control's state can be checked for with the help of the Elvis (?) operator.

```
<div *ngIf="form.controls.firstName?.valid &&
  form.controls.firstName?.touched" class="alert
  alert-danger">
  First Name is required
</div>
```

contact-form2.component.html

Model driven forms

- Model-based approach of writing Forms is also known as *reactive approach*.
- To use the reactive approach one needs to import `ReactiveFormsModule` from `@angular/forms` as opposed to the `FormsModule`.

```
import { ReactiveFormsModule } from '@angular/forms';
…
@NgModule({
  imports: [ ..., ReactiveFormsModule ],
  declarations: [ ... ],
  bootstrap: [ ... ]
})
export class AppModule {}
```

Model driven forms

- Controls and groups are programmatically constructed using the `FormControl` and `FormGroup` classes.

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

@Component({
  ...
})
export class App {
  form = new FormGroup({
    firstname: new FormControl(),
    lastname: new FormControl(),
  address: new FormGroup({
    street: new FormControl(),
    zipCode: new FormControl()
   })
 });
}
```

Model driven forms...

- The programmatically constructed form needs to be hooked to the template using the `[formGroup]` binding.

```
<form [formGroup]="form">
        ...
</form>
```

- The group and form controls are mapped using `formControlName` and `formGroupName`.

contact-formapp.component.ts
contact-form1.component.ts
contact-form1.component.html
contact-form1app.module.ts

Validators

- Angular provides a set of built-in validators out of the box.
- Some of them include
  - required
  - minlength
  - maxlength
  - pattern
  - email

```
form = new FormGroup({
    firstname: new
  FormControl('',Validators.required),
    lastname: new FormControl('',
      Validators.compose([Validators.required,
  Validators.minLength(2)])),
});
```

Custom validators

- In its simplest form a validator is essentially a function that takes a `Control` and returns either
  - `null`, when valid
  - `error` object, if invalid

```
function zipValidator(zip:any){
        var valid = /^\d{6}$/.test(zip.value);
        if (valid)
            return null;
        else
            return {valid: false};
 }
```

contact-form3.component.ts
contact-form3.component.html

Under the hood

Lifecycle hooks

- Directives and component instances have a lifecycle when Angular creates, updates and destroys them.
- Angular
  - creates components, renders them,
  - creates and renders their children,
  - checks when data-bound properties change and
  - destroys them before removing from the DOM
- Angular offers ***lifecycle hooks*** that provide us access to these key life cycle moments and allow us to act when they take place.
- A directive has the same set of hooks as a component except for the ones related to content and views.

Lifecycle Sequence

Hook	Purpose
ngOnChanges	When Angular changes data-bound input properties
ngOnInit	Initialize the directive/component after Angular first displays the data-bound properties
ngDoCheck	Detect and act upon changes that Angular may not detect on its own
ngAfterContentInit	After Angular projects external content into the component's view
ngAfterContentChecked	After Angular checks the content projected into the component
ngAfterViewInit	After Angular initializes the component's and the child views
ngAfterViewChecked	After Angular checks component's and child views
ngOnDestroy	Cleanup just before Angular destroys the directive/ component

Lifecycle hooks

- Developers can tap into the lifecycle by implementing one or more of the hook interfaces in the Angular **`core`** library.
- Every interface has a single hook method named after the interface name prefixed with **`ng`**.
- **`OnInit`** interface has a hook named **`ngOnInit`** called by Angular after creating the component.

OnInit

- **ngOnInit** is traditionally used for:
  - Performing complex initializations after construction
  - Setting up the component after Angular sets the input properties.
- **ngOnInit** is a good place for a component to fetch its initial data.
- Constructors should simply be confined to set the initial local variables to simple values.

OnDestroy

- Cleanup logic is placed in **`ngOnDestroy`**, the logic that must get processed before Angular destroys the directive.
- This is the section to free resources that will not be garbage collected automatically.
- Unsubscribe from DOM events, stop interval timers, unregister callbacks as applicable.
- This is the place to notify the other part of the application that the component will go away.

lifecycleHooksDemo.component.ts
hooksDemoapp.component.ts

Services

Services

- If some code is needed by multiple components in an application, it is a a good practice to create a single reusable service.
- When a component needs this service, it can be injected.
- *A service is a mechanism used to share functionality over components*.
- Service is the ideal approach to help bring external data into an application.
- Service can be shared between multiple components.
- Any logic not encapsulated in a view should be segregated into a service.

Create a Service

- A service in Angular is a class.
- This class needs to be marked as a service which can be done as follows
- Declare a decorator called **`@Injectable()`**
  - This is used to make the service class injectable in other components in the application.

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

Demo1.service.ts

Service

- The consumer of the service need not be aware of how the service obtains the data.
  - The data could be retrieved from a web service or local storage
- This helps separate the data access logic from the component.
- In order to use the service, it requires to be imported.
- In the constructor of the component, the service is injected.
- The service is also added to the **`Providers`** property of the component's **`metadata`**.

Demo1Serviceapp.component.ts

Services...

- Where are factories, services, constants and values?
- They’re gone!
- A service is the mechanism used to share functionalities over Components.
- Components may need to manipulate data.
- In the real world, data comes from the server
- Service represents the ideal location to retrieve external data into an application
- Service can be shared between many components.
- Services can also be used to share functionalities across components in an application.

Singleton

- In Angular services are singleton per provider.
- In case the service is provided multiple times, multiple instances are provided.
- If provided in

```
@NgModule({
    Providers: [service]
    …
})
```

- This will be provided in the root scope and a single instance is provided for the entire application.
- If a service is provided in a **`@Component()`** then every component instance and its children will get a service instance.

Meta Service

- A service has been introduced to easily retrieve or update meta tags.

```
import {Meta} from '@angular/platform-browser';

constructor(meta: Meta){
  meta.addTag({name: 'author', content:
            'Lakshman M N'});
  meta.addTag({name: 'description',
      content: 'Awesome stuff'});
}
```

Demo2Serviceapp.component.ts

HTTP

HTTP

- Most angular applications obtain data using HTTP.
- The application issues a GET request to a web service.
- The web service retrieves the data often using a data store and returns it to the application using a HTTP response.
- The application subsequently processes the said data.
- HTTP in Angular employs the concept of *Observables*.

Observables

- An observable can be thought of as an array whose items arrive asynchronously over time.
- Observables help manage asynchronous data such as data coming in from a backend service.
- Observables are a proposed feature for ES 2016.
- Observables derive their name from the Observer design pattern.
- In order to use Observables now Angular uses a third party library called Reactive Extensions (RxJS)
  - This is not to be mixed up with reactJS.
- Observables are used in Angular itself in the event system and the HTTP client service.

Promises

- A Promise represents the eventual result of an operation.
- Promise can be used to specify what to do when an operation eventually succeeds or fails.
- Promises execute asynchronous functions in series by registering them with a promise object.
- ES6 specification added native support for Promises.
  - Therefore Promises can be used in any ES6 code and not specific to Angular.

Promise vs Observable

Promise	Observable
Returns a single value	Works with multiple values over time
Not cancellable	Cancellable
Can not be retried	Can be retried

Angular 1 vs Angular 2.x +

- Angular 1 used **`$http`** that returned promises.
  - **`$http.get`**
  - **`$http.post`**
- Angular Http returns “observables” through RxJS (Reactive Extensions).
  - Promises are available too
- Observable streams provides flexibility in the context of handling responses from HTTP requests.

Using the HTTP client

- Services typically could provide access to data either locally using local storage or a remote data store.
- In order to access server based resources we need the Angular **`Http`** package.
  - It will most likely be declared as a dependency in **`package.json`**
  - Hence it would be installed in the project by **`npm`**
- This allows us to import the **`Http`** class in our service.

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

- The **`Http`** service is employed to make HTTP requests.

Http Service

- In order to use the **Http** service we add a constructor in our service that accepts a parameter of type **Http**.
- Angular needs to be told to perform dependency injection on this service.
- Our service needs to be decorated as **Injectable**.

```
import { Injectable } from '@angular/core';
@Injectable()
export class StatsService {
  constructor(private http: Http){
  }
}
```

Http Service

- We don't need the **`@Injectable()`** in the component classes as they already have a **`@Component()`** decorator.
- On plain classes in order for Angular to know that we intend to inject dependencies we need the **`@Injectable()`** decorator.
- The next step is to tell Angular about the **`HttpModule`** in **`app.module`**
- The **`Http`** service is part of the **`HttpModule`**.
  - This provides not just the **`Http`** service but all other services that are in turn required by the **`Http`** service.

app.module.ts

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

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

statsapp.module.ts

`Http` Service

- The **`http`** object provides various methods basically one for each kind of HTTP request.
- The **`http.get()`** returns a Observable of a Response.
  - There exists a Response class with features like status and data.
- Since the HTTP request is a asynchronous operation, the **`get()`** method may not return a Response immediately.
- The Observable can be used to be notified when a Response is available.
  - Observable is from the ***RxJS*** library.

Using `Http` Service

- Out of the box an Observable has a limited number of methods.

```
import 'rxjs/Rx';
```

- Importing this in our service provides a lot more functionality.
  - This loads all the possible operators/methods of an Observable.
- One of the methods is **`toPromise()`** that converts an Observable into a Promise.
- When the Response is available it will be passed to the function specified using the **`then()`** method.
- The **`json()`** method on the response parses the response as JSON.

Using `Http` Service

- The method in our service returns a Promise of `Stats` that indicates that `Stats` will be returned but not immediately, instead when there is a response from the HTTP server.
- The component using the service uses the **`then()`** method on the returned Promise to access the data from the JSON response.
- The Elvis operator can be used to tell Angular to retrieve a property value only when available.

stats.model.ts
stats.json
statsapp.component.ts
stats.service.ts

Using `Http` Service

- Search parameters can be added to an `HTTP` request as follows:

```
http.get(url, { params: { sort: 'descending' } });
```

- Previously the same would have been as follows:

```
const params = new URLSearchParams();
params.append('sort', 'descending');
http.get(url, { search: params })
```

Http Service

- Employ the **`Http`** service to communicate with REST API.
- ***JSONPlaceholder*** is a free online REST service
  - http://jsonplaceholder.typicode.com

```
url: string = 'http://
  jsonplaceholder.typicode.com/posts'
getPosts(): Promise<Posts> {
        return this.http.get(this.url).toPromise()
            .then(response => response.json())
}
```

posts.model.ts
postsapp.component.ts
posts.service.ts

HTTP Module & Observables

```
getPosts() {
    this.http.get('/app/posts.json')
      .map((res:Response) => res.json())

      .subscribe(
         (data) => { this.foods = data},  //onNext
         (err) => console.error(err),          //onError
         () => console.log('done')       //onCompleted
      );
  }
```

`.subscribe()` Event Handlers

`onNext`: Receives http response data as an observable. Support streams of data and can call this event handler multiple times

`onError`: If http requests returns error like 404

`onCompleted`: Executes after the observable is finished returning data

posts.model.ts
posts1app.component.ts
posts1.service.ts

Http Service - POST

- The **post()** method on the Http service accepts a few parameters:
  - **url** as string
  - **body** as object
  - **options** (optional) as **RequestOptions**
- It returns an Observable of type **Response**.

```
getPosts() {
        return this.http.post(this.url, {
      'userid':1,
      'title':'Hello World',
      'body':'This is a new post!'})
        .map((responseData) => {
            return responseData.json()
        })
  }
```

posts.model.ts
posts2app.component.ts
posts2.service.ts

Http Service - PUT

- The **put()** method on the Http service has a similar set of parameters as the **post()**.

```
url: string = 'http://jsonplaceholder.typicode.com/posts/
  1';
getPosts() {
        return this.http.put(this.url, {
      'id': 1,
      'userid':1,
      'title':'Hello World',
      'body':'This is a new post!'})
        .map((responseData) => {
            return responseData.json()
        })
    }
```

posts.model.ts
posts3app.component.ts
posts3.service.ts

Http Service - DELETE

- The **delete()** method on the Http service accepts the URL and RequestOptions (optional).

```
url: string = 'http://jsonplaceholder.typicode.com/
  posts/1';
getPosts() {
        return this.http.delete(this.url)
        .map((responseData) => {
            return responseData.json()
        })
    }
```

posts.model.ts
posts4app.component.ts
posts4.service.ts

Summary

- HTTP is one of the protocols used to allow a browser to communicate with the server and vice-versa.
- `Http` is a service available to be imported in Angular.
- In Angular when a method like GET / POST on the Http service is invoked the return is an observable.
  - We can transform this into a promise if necessary.
- This observable needs to be *subscribed* to in order to access the data.

Routing

Routing and Navigation

- Navigating from one page view to another is crucial in a single page application (SPA).
- Screens seen by the user during navigation through the app need to be managed.
- Angular routes enable the creation of different URLs for different content in an application.
- Having different URLs for different content allows users to bookmark URLs to specific content.
- **Each bookmarkable URL in Angular is called a route.**

Overview

- The *Angular Router* enables navigation from one view to another based on tasks performed by users.
- The Router can regard a browser URL as an instruction to navigate to a client generated view.
  - Optional parameters can be passed to the view component to help decide the content to be presented.
- The Router can be bound to links on a page and will navigate to the appropriate application view when the link is clicked upon by the user.

Angular Router

- Angular Router is an optional service that helps present a particular component view for a given URL.
  - It is not a part of the Angular core.
- It resides in its own library package, **`@angular/router`**
- This library package requires to be imported.

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

- An application will have one *router*.
- When the browser URL changes, the router looks for a corresponding *Route* to determine the component to display.
- *A router has no route definitions unless configured.*

Angular Router

- The **`RouterModule.forRoot()`** method is used to provide an array of routes to specify navigation.
- Each *Route* maps a URL path to a component.

```
RouterModule.forRoot([
      { path: 'animals', component: AnimalsComponent },
      { path: 'birds', component: BirdsComponent },
      { path: '', component: HomeComponent },
      { path: '**', component: PageNotFoundComponent }
  ])
```

- There are no *leading slashes* in the *path*.
  - The router parses and builds the URL.
- The empty path matches as the default path for each level of routing.
- The '**' indicates a wildcard path for the route.
  - The router matches this route if the URL requested does not match any paths or routes defined.

Angular Router

- **Router Outlet**

```
<router-outlet></router-outlet>
```

  - The *RouterOutlet* is used to display views for a given route.
  - This is where templates of specific routes are loaded as we navigate.
- **Router Links**

```
<a routerLink="./Birds">Birds</a>
```

- Most often than not navigation is a result of user action such as click of a link.
- The *RouterLink* directive is added to the anchor tag.
  - The *RouterLink* directive substitutes the normal **`href`** property and makes it easier to work with route links in Angular.

app.routes.ts
home.component.ts, birds.component.ts, animals.component.ts
Routerindex.html, Routerapp.module.ts, routerapp.component.ts

Summary

- The Angular Router helps navigate from one view to another.
- Routing helps divide an application into logical views and bring up different component views as appropriate.
- Single Page Applications can be built in a straight-forward manner using Angular capabilities.

Testing

Unit Testing

- Unit is source code that are small individual testable parts of an application.
- Unit testing represents the software process in which these small units of code are tested to ensure their appropriate operation.
- Unit tests are short code fragments created by programmers during the development process.

Getting Started

- Testing in Angular is deeply integrated in the framework with the **`angular/testing`** package.
- In Angular Components are directly exported classes and usually have dependencies – service or pipes.
  - Angular 1 required to manually stub these instances out and provide mocks for them
  - Angular removes the need for us to bootstrap the applications when writing our tests.
  - Tests can be written like calling any method and the component's properties can be checked for.

Angular Testing Tools

- *Jasmine*
  - Jasmine allows us to write tests to ensure our applications are behaving the way we expect them to.
  - It is a Behaviour-Driven Development (BDD) framework built for testing JavaScript code.
    - http://jasmine.github.io/2.5/introduction.html
  - By convention tests written in Jasmine are called `specs`
- *Karma*
  - A test runner provided by the Angular team, Karma executes tests in multiple browsers thereby ensuring application compatibility across browsers.

Jasmine

- Jasmine `specs` are a description of a feature or a unit of code.
- Specs usually start with a *describe* block that contains tests connected with that feature.

```
describe("MetricConverter", function(){
                                    });
```

- The first argument gives a short description to the tested feature, the second argument is a function that executes its expectations.

Jasmine

- Jasmine's **`it`** block is used to set up and test response.

```
it("returns status code", function(){ })
```

- Jasmine implements expectations with the **`expect()`** function that tests if a tested object matches some of the expected conditions.

```
expect(response.statusCode).toBe(200);
```

First Tests

- **app.ts**

```
import {Component} from 'angular/core';
@Component({
  selector: 'app',
  template: '<span>{{hello}}</span>'
})
export class App {
  private hello: string = 'Hello World!';
}
```

TestApp.ts

First Tests

- **app.spec.ts**

```
import {App} from './app';
describe('App', () => {
  beforeEach(function() {
    this.app = new App();
  });

  it('should have hello property', function() {
    expect(this.app.hello).toBe('Hello World');
  });
});
```

TestApp.spec.ts
npm test

Build and Deployment

Bundling

- **`ng build --prod --aot`**
  - run in commandline when the directory is **`projectFolder`**
  - flag **`prod`** bundle for production and flag **`aot`** enable the ahead-of-time compilation also known as offline compilation
  - bundles are generated by default to **`projectFolder/dist`**

Output

- **`dist/main.[hash].bundle.js`**
  - Your application bundled
- **`dist/vendor.[hash].bundle.js`**
  - Your dependencies bundled (@angular, RxJS etc)
- **`dist/polyfill.[hash].bundle.js`**
  - Polyfill dependencies
- **`dist/index.html`**
  - Entry point of your application
- **`dist/style.[hash].bundle.css`**
  - Style definitions

Session Summary

- Angular is a framework for building JavaScript-rich web applications.
  - Used to build a SPA.
- Angular provides applications a loosely coupled architecture.
  - This helps maintainability of the application.
- Angular provides a lot of features that help develop applications faster with less code.
  - This improves productivity
- Angular enjoys a great community support!

Resources

- Links
  - Upgrading Apps to Angular
    - http://blog.thoughtram.io/angular/2015/10/24/upgrading-apps-to-angular-2-using-ngupgrade.html
  - Rangle.io - Introduction to Angular 2 Course
    - http://info.rangle.io/angular-2-online-training
  - TypeScript in 30 minutes
    - http://tutorialzine.com/2016/07/learn-typescript-in-30-minutes/
  - Built with Angular
    - http://builtwithangular2.com
- Books
  - Angular Book - Sebastian Eschweiler
  - Learning Angular - Pablo Deeleman