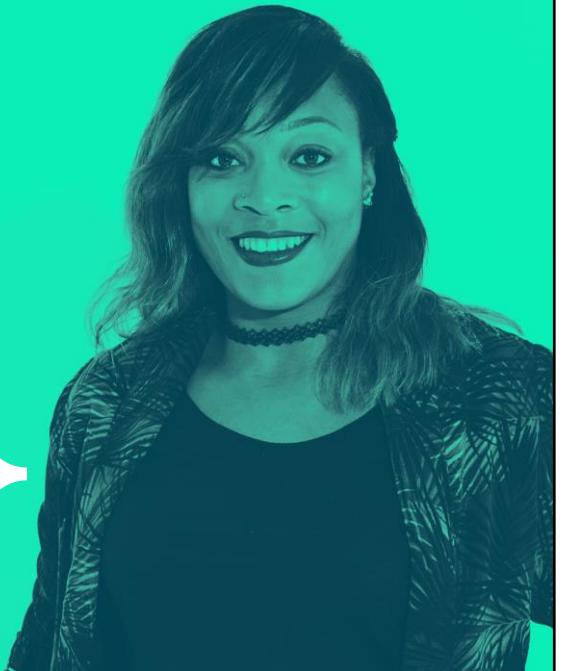




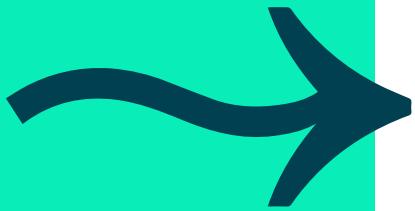
# Welcome!

Building Web Applications using Angular





## Overview



### Objectives

- To explain the aims and objectives of the course

### Contents

- Course administration
- Course objectives and assumptions
- Introductions
- Any questions?

### Exercises

- Locate the exercises
- Locate the help files



## Administration



Front door security	Downloads and viruses
Name card	Admin. support
Chairs	Messages
Fire exits	Taxis
Toilets	Trains/Coaches
Coffee Room	Hotels
Timing	First Aid
Breaks	
Lunch	Telephones/Mobiles

We need to deal with practical matters right at the beginning.

Above all, please ask if you have any problems regarding the course or practical arrangements. If we know early on that something is wrong, we have the chance to fix it. If you tell us after the course, it's too late! We ask you to fill in an evaluation form at the end of the course. If you alert us to a problem for the first time on the feedback form at the end of the course, we won't have had the opportunity to put it right.

If this course is being held at your company's site, much of this will not apply or will be outside our control.



## Course delivery



Hear and Forget  
See and Remember  
Do and Understand



The course will be made up of lecture material coupled with the course workbook, informal questions and exercises, and structured practical sessions. Together, these different teaching techniques will help you absorb and understand the material in the most effective way.

The course notebooks contain all the overhead foils that will be shown, so you do not need to copy them. In addition, there are extra textual comments (like these) below the foils, which are there to amplify the foils and provide further information. Hopefully, these notes mean you will not need to write too much and can listen and observe during the lectures. There is, however, space to make your own annotations too.

The appendices cover material that is beyond the scope of the course, together with some help and guidelines. There are also appendices on bibliography and Internet resources to help you find more information after the course.

In the practical exercise sessions, you will be given the opportunity to experiment and consolidate what has been taught during the lecture sessions. Please tell the instructor if you are having difficulty in these sessions. It is sometimes difficult to see that someone is struggling, so please be direct.



## The training experience

A course should be

- A two-way process
- A group process
- An individual experience



The best courses are not those where the instructor spends all of his/her time pontificating at the front of the class. Things get more interesting if there is dialogue, so please feel free to make comments or ask questions. At the same time, the instructor has to think of the whole group, so if you have many queries, you might be asked to deal with them off-line.

Work with other people during practical exercise sessions. The person next to you may have the answer, or you may know the remedy for them. Obviously do not simply 'copy from' or 'jump-in on' your neighbour, but group collaboration can help with the enjoyment of a course.

We are also individuals. We work at different paces and may have special interests in particular topics. The aim of the course is to provide a broad picture for all. Do not be dismayed if you do not appear to complete exercises as fast as the next person. The practical exercises are there to give plenty of practical opportunities; they do not have to be finished and you may even choose to focus for a long period on the topic that most interests you. Indeed, there will be parts labelled 'if time allows' that you may wish to save until later to give yourself time to read and absorb the course notes. If you have finished early, there is a great deal to investigate. Such "hacking" time is valuable. You may not get the opportunity to do it back in the office!



## Course Outcomes

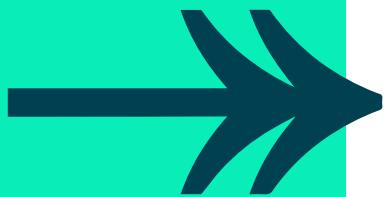
By the end of the course, you will be able to:

- Explain what the building blocks of Angular are
- Create a Single Page Application that:
  - Uses a number of components with data binding
  - Uses custom directives and pipes
  - Leverages the power of RxJS Observables
  - Uses Services to handle data
  - Is fully routed
  - Is tested





## Assumptions



### This course assumes the following prerequisites

- HTML and CSS skills equivalent to those provided by the Web Development Fundamentals – HTML and CSS course
- JavaScript skills equivalent to those provided by the Web Development Fundamentals – JavaScript course
- Familiarity with TypeScript, equivalent to that provided by the Programming with TypeScript course

*If there are any issues, please tell your instructor now*

**If you are not sure of any of these, please inform the instructor as soon as you can and they will do their best to help you.**



## Introductions

**Please say a few words about yourself**

**What is your name and job?**

**What is your current experience of**

→ Computing?

→ Programming?

→ Web development?

**What is your main objective for attending the course?**



One of the great benefits of courses is meeting other people. They may have similar interests, have encountered similar problems and may even have found the solution to yours. The contacts made on the course can be very useful.

It is useful for us all to be aware of levels of experience. It will help the instructor judge the level of depth to go into and the analogies to make to help you understand a topic. People in the group may have specialised experience that will be helpful to others.

It is worth highlighting particular interests, as we may be able to address them during the course. However, it is a general course that aims to cover a broad range of topics, so the instructor may have to deal with some areas during a coffee break or over lunch.



## Any questions

### Golden Rule

→ "There is no such thing as a stupid question"

### First amendment to the Golden Rule

→ "... even when asked by an instructor"

### Corollary to the Golden Rule

→ "A question never resides in a single mind"



Please feel free to ask questions.

Teaching is a much more enjoyable and productive process if it is interactive. You will no doubt think of questions during the course. If so, ask them!



# Introduction

Building Web Applications using Angular



## Objectives

- To become aware of what Angular is
- To understand the difference between Angular and AngularJS
- To be aware of the versioning and release schedules for Angular
- To be able to set up the developer environment using Angular CLI
- To become familiar with some fundamental Angular CLI commands
- To be aware of developer tools available for Angular



Angular Logo from: <https://angular.io/presskit>

## What is Angular?

A framework for building scalable web applications

Built and maintained by Google

Angular, aka Angular 2, aka Angular N\* is the latest framework from the team that brought you AngularJS aka Angular 1

Significantly different from AngularJS

- No \$scope
- No controllers
- Uses TypeScript

N = Any number greater than or equal to 2

Why the big difference between Angular and AngularJS?

- AngularJS has been around since 2009, before ECMAScript 5 and HTML5
- The face of web applications and the technology powering them has changed significantly and at an accelerating pace since AngularJS
- Introduction of transpilation has promoted the ability to build larger, more complex web applications

## Semver

Semantic versioning defines the versioning of a software project and should follow the format X.Y.Z where:

- Z is the patch version, a change here indicates there are no breaking changes – just bug fixes
- Y is the minor version, a change here indicates new features have been introduced but again, no breaking changes
- X is the major version, a change here indicates breaking changes have been introduced and this version is no longer compatible with the previous

Starting with Angular 2, Angular follows Semver along with a release schedule

- A new patch every week
- Three monthly minor releases
- One major release every six months

What happened to Angular 3 then? (Skipped in order to bring versions in line with Router)

## The Developer Environment – Angular CLI

Angular provides a CLI package – available from NPM

Provides:

- Skeleton applications
- Creation, scaffolding and integration of various Angular building blocks
- Support for building and testing applications

Should be installed globally on developer's system:

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

Provides access to **ng** executable that can be followed by a command and options

- Creating Angular building blocks
- Running tests
- Upgrading versions

# Using the CLI

The **new** command can be used to create a skeleton Angular application

```
ng new <path/and/nameOfApp> [options]
```

- Command can be run directly from folder to contain application or by supplying path in name

The **generate** command can be used to create new items such as Components and Modules

```
ng generate <schematic> [options]
```

- Some of the 14 schematics that can be created using this command will automatically be integrated for use in the application, others will have to be configured manually

# Using the CLI

Other notable commands:

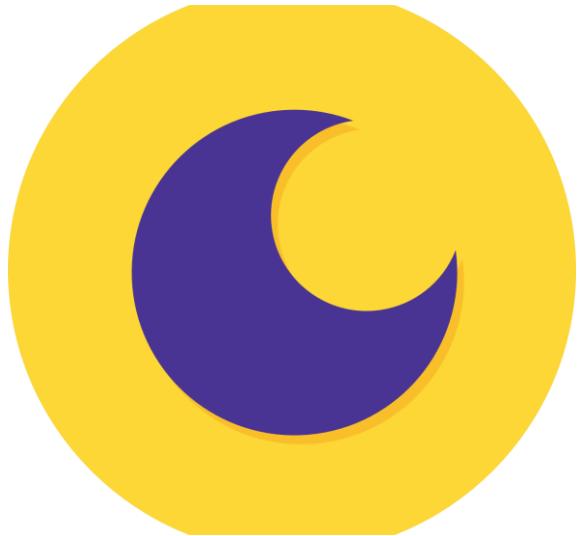
Command Syntax	Description
<code>ng doc &lt;keyword&gt; [options]</code>	Opens official Angular documentation for given keyword
<code>ng help [options]</code>	Lists available commands and their short descriptions
<code>ng update [options]</code>	Updates the application and its dependencies
<code>ng add &lt;collection&gt;</code>	Add an <b>npm</b> package for a published library to your workspace
<code>ng serve &lt;project&gt; [options]</code>	Builds and serves the app, rebuilding on file changes
<code>ng test &lt;project&gt; [options]</code>	Runs unit tests defined in a project
<code>ng build &lt;project&gt; [options]</code>	Compiles an Angular application into folder <b>/dist</b> at given path

For more see <https://angular.io/cli>

## **Augury – Debugger and Profiler for Angular**

Open-source add-on that can be installed in Chrome or Firefox

- Maintained by Google and rangle.io
- Helps developers visualize an Angular application through component trees and debugging tools
- Insight into application structure, change detection and performance characteristics



Logo from: <https://augury.rangle.io/>



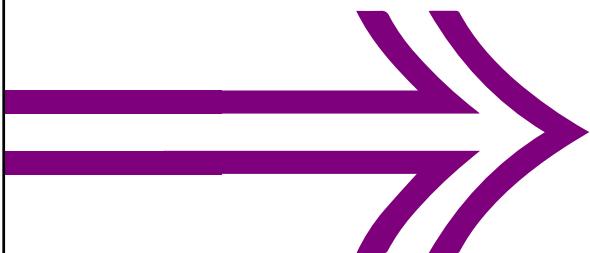
## Objectives

- To become aware of what Angular is
- To understand the difference between Angular and AngularJS
- To be aware of the versioning and release schedules for Angular
- To be able to set up the developer environment using Angular CLI
- To become familiar with some fundamental Angular CLI commands
- To be aware of developer tools available for Angular



Angular Logo from: <https://angular.io/presskit>

## **QuickLab 1 – CLI and Augury**



In this QuickLab you will:

- Set up and use the Angular CLI
- Set up a new Angular application and run it in the browser
- Install Augury in the browser and examine the output



# Angular Architecture

Building Web Applications using Angular



## Objectives

- To be aware of the eight building blocks of Angular

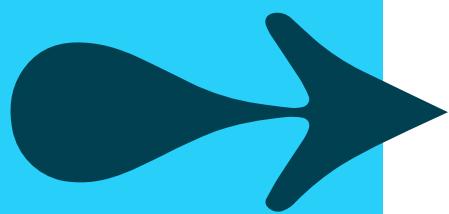
Modules  
Components  
Templates  
Metadata  
Data binding  
Directives  
Services  
Dependency injection



Angular Logo from: <https://angular.io/presskit>



## ANGULAR OVERVIEW



Platform and framework for building client applications in HTML and JavaScript (via TypeScript)

Most basic building block are **NgModules**

- Provide compilation context for components
- Collect related code into function sets
- Angular application is defined by a set of **NgModules**
- App always has at least a **root** module
- Enables bootstrapping and typically has many more **feature** modules

# Angular Overview

**NgModules** usually have at least one **Component**

**Components:**

- Define **VIEWS** – sets of screens that Angular can choose from and modify according to program logic and data

- Use **Services** – provision of business logic not related to **VIEWS**

Can be injected into components as dependencies

Allows modularity and code reuse

**NgModules, Components** and **Services** are classes with *Decorators* to define type and **Metadata**

- **NgModule Metadata** declares **Components**, imports other modules, exports symbols **and** provides **Services** – root module bootstraps first view Component

- **Component Metadata** links a **Template** and **Styles** to define a **VIEW** and where to inserted it

Ordinary HTML combined with Angular **Directives** and **Data Binding** to allow Angular to modify **VIEW**

- **Service Metadata** tells Angular how to make it available to components through **Dependency Injection**

# Modules (Angular modules/NgModules)

## **NOTE – THESE DIFFER TO ES2015 JAVASCRIPT MODULES**

Angular is modular, beginning with the *root module*, often called  **AppModule**

Most applications will have many feature-based modules, encapsulating related code

Takes form as a class with an **@NgModule** decorator with **Metadata** that describes:

Metadata	Description
<b>declarations</b>	the view classes that this module declares (components, directives and pipes)
<b>exports</b>	the declarations that are visible to other modules
<b>imports</b>	modules whose exported classes are needed in this module
<b>providers</b>	creators of services that this module adds to the global collection of services
<b>bootstrap</b>	only set by the root module, this declares the main application view

# Modules

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
// Note these are ES2015 JavaScript Module imports!
import { AppComponent } from './src/AppComponent';
import { SomeService } from './src/SomeService';

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

ES2015 JavaScript Module import are an important part of using Angular – they are needed to expose code from other files.

Angular Modules are different to these!

# Components

**Components** control parts of the *VIEW*

A **Component** is encapsulated within a single **Module**

- Is a class with a **@Component Decorator** which contains the **Metadata** for:
    - selector** – the HTML-like element to insert the component at
    - template** or **templateUrl** - An HTML **Template** (either as a string or a file location)
    - Styles** or **StylesUrl** – An string of styles or an array of CSS style sheets to be applied to this component
    - providers** – An array of providers for services that the component requires
- The class interacts with the *VIEW* through an API of properties and methods
- Angular creates, populates and destroys **Components** as the user interacts with the application

# Components

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: []
})
export class AppComponent {
  title = 'app works!';
}
```

## Templates

Templates are used to tell Angular how to render the component  
They are written in normal HTML with some additions from Angular

```
<h1>  
  {{title}}  
</h1>
```

# Metadata

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}

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

Did you notice how modules and components are just classes?

Metadata is how we tell Angular what this class is

In TypeScript we attach this metadata through the use of [decorators](#)

For both our module and component decorators we pass a required configuration object

# Data binding

Data binding is the mechanism through which we link parts of a component with parts of the template

Depending on the form we use, we can send data

- From the component to the template
- From the template to the component
- In both directions!

```
<h1>
  {{title}}
</h1>
```

## Directives

Components are a type of directive, a directive with a template

The remaining two types of directives are structural and attribute

- Structural directives alter layout through adding and/or removing DOM elements
- Attribute directives alter layout or behaviour of existing DOM elements

There are built-in directives but we can also write our own (like Components!)

# Services

Component's job is to bridge between the view and the application logic

All non-trivial tasks should be carried out by services

A service is *typically* a class but could actually be anything

- In general, it should have a narrow, well-defined purpose

```
export class Logger {  
  log(thing: any) { console.log(thing); }  
  error(thing: any) { console.error(thing); }  
}
```

Services can be injected into Components, giving access to their functionality

- This is not enforced rather made easier by Dependency Injection

# Dependency Injection

Modules, components, directives, services – with this many building blocks floating around things will get confusing with dependencies upon dependencies

Angular handles dependencies using Dependency Injection (DI)

The injector holds instances of services it has previously created, and should a requested service not be within then it creates a new instance

Typically we are injecting services into components by passing them in as parameters to the component's constructor function

We register providers with the injector, in this case the service class itself

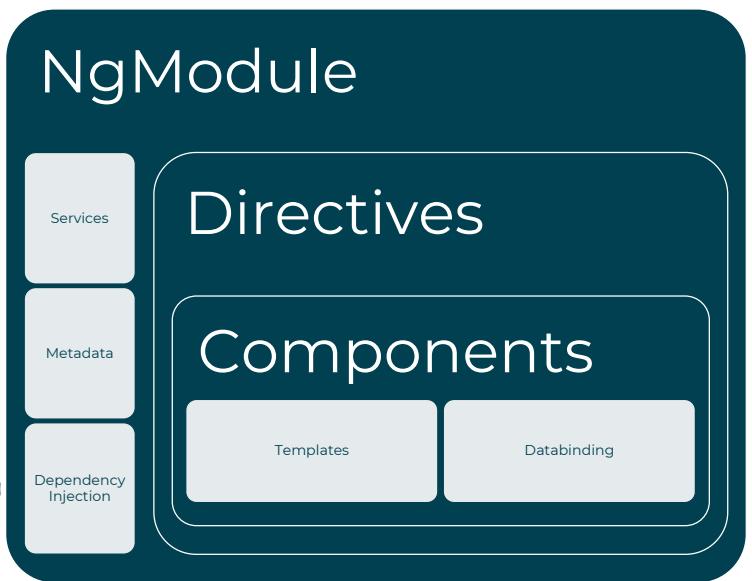
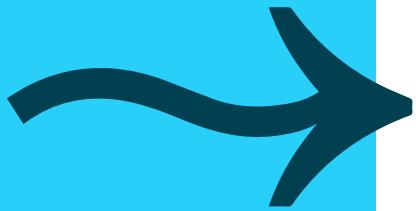
If you register them in a module they are available for that module – generally it's a good idea to register them with the root module so the same service instance is available everywhere

- Can be done in the Service using the Injectable decorator and a provided In key with the module name – usually root
- Can be done in the NgModule the Service is used in using the provider key in metadata - for application scope, register it with the root module

If you register a provider with a component then a new instance is created with each new instance of the component



**BRING THEM  
ALL  
TOGETHER**



# Angular Overview

**NgModules** usually have at least one **Component**

**Components:**

- Define **VIEWS** – sets of screens that Angular can choose from and modify according to program logic and data

- Use **Services** – provision of business logic not related to **VIEWS**

Can be injected into components as dependencies

Allows modularity and code reuse

**NgModules, Components** and **Services** are classes with *Decorators* to define type and **Metadata**

- **NgModule Metadata** declares **Components**, imports other modules, exports symbols **and** provides **Services** – root module bootstraps first view Component

- **Component Metadata** links a **Template** and **Styles** to define a **VIEW** and where to inserted it

Ordinary HTML combined with Angular **Directives** and **Data Binding** to allow Angular to modify **VIEW**

- **Service Metadata** tells Angular how to make it available to components through **Dependency Injection**

## QuickLab 2 – Angular Architecture



In this QuickLab you will:

- Examine the folder structure of an Angular project set up by the CLI
- Explore key files and building blocks
- Use the CLI's built-in documentation functionality
- View and change the application



## Objectives

- To be aware of the eight building blocks of Angular

Modules  
Components  
Templates  
Metadata  
Data binding  
Directives  
Services  
Dependency injection



Angular Logo from: <https://angular.io/presskit>



# Testing with Jasmine

Building Web Applications using Angular





## Objectives

- To understand how a Jasmine Unit Test is written
- To be able to use spies in tests
- To be able to use functions to set up and tear down tests
- To understand how asynchronous testing can be done
- To be able to debug Jasmine tests
- To be able to generate code coverage reports from Jasmine tests



Angular Logo from: <https://angular.io/presskit>

Jasmine Logo from: <https://jasmine.github.io/>

# Testing

Integral part of software development

Angular CLI projects come ready made to test:

- Jasmine – “Jasmine is a behaviour-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests”
- Karma – A test runner written by the AngularJS team, Karma is well suited for integrating tests into the development process
- Angular testing utilities – A number of tools used to create a controllable environment in which we can reliably test out code as it interacts with Angular

If the CLI is not used, testing has to be set up manually

## Jasmine Overview

Open Source JavaScript framework capable of testing **any kind** of JavaScript application  
Follows Behaviour-Driven Development to ensure that each line of JS is properly unit tested  
Provides small syntax to test smallest unit of entire application instead of testing it as a whole

Features:

- Doesn't depend on any other JavaScript Framework
- Doesn't require any DOM
- Clean and obvious syntax

# Jasmine Test Building Blocks

## SUITES

Basic building block of Jasmine Framework

Collections of similar type test cases written for a specific file or function

Can be nested

Usually contain a `describe` function and at least 1 call to the `it` function

### `describe()`

Call to this function requires a string, used to identify the suite and a function that contains calls to the `it` functions

### `it()`

Calls to this function require a string, used to identify the test case it represents and a function that defines how the test will be evaluated – through calls to the `expect` function

# Jasmine Test Building Blocks

## `expect()`

Defines the **actual value** obtained by running the test case

- `it()` calls can have more than 1 `expect()` call

Uses **matchers** to define how to compare the expected result with the actual

## **MATCHERS**

Jasmine provides a host of in-built matchers

Each does a Boolean comparison of the actual and expected outputs

Custom matchers can be written for cases where in-built do not meet the requirements

# Jasmine MATCHERS

Matcher	Description
<code>.nothing()</code>	Expect nothing explicitly
<code>.toBe(expected)</code> , <code>.toEqual(expected)</code>	Expect the actual value to be === to the expected, for toEqual, objects and their properties can be compared
<code>.toBeFalsy()</code> , <code>.toBeNull()</code> , <code>.toBeUndefined()</code>	Expect the actual value to be Falsy, explicitly null or explicitly undefined
<code>.toContain(expected)</code>	Expect the actual value to contain the expected value – can be in an array as well as a substring in a string
<code>.toBeGreaterThan(expected)</code> , <code>.toBeGreaterThanOrEqual(exp)</code>	Expect the actual value to be greater than (or equal to) the expected value
<code>.toBeLessThan(expected)</code> , <code>.toBeLessThanOrEqual(exp)</code>	Expect the actual value to be less than (or equal to) the expected value
<code>.not.</code>	Invert the result of the matcher (placed after expect and before matcher)

## Example Spec

```
describe(`A suite is just a function`, () => {
  let a;
  let b;
  it(`and so is a spec`, () => {
    a = true;
    b = `Another string`;
    expect(a).toBe(true);
    expect(a).not.toBeFalsy();
    expect(b).toContain(`her`);
    expect(b).not.toMatch(/foobar$/);
  });
});
```

For this test to report as **PASSED**, all `expect` calls must return true.

**NOTE:** This is not an ideal way to test these situations!

## Custom Matchers

Sometimes the built-in matchers will not fit with the test required

Jasmine allows custom matchers to be written

- As long as they follow the correct pattern

Jasmine's documentation explains how to create these:

[https://jasmine.github.io/tutorials/custom\\_matcher](https://jasmine.github.io/tutorials/custom_matcher)

## Jasmine Spy

- Allows spying on the test code's function calls
- Intercepts call to actual function and allows logging and monitoring of calls to it
- For existing functions, the `spyOn()` method needs to receive the object and the name of the function

Jasmine provides a number of matchers to verify that the function has been used appropriately:

Matcher	Description
<code>toHaveBeenCalled()</code>	Expect the function spied on to have been called at least once
<code>toHaveBeenCalledBefore(exp)</code>	Expect the function spied on to have been called before another spy
<code>toHaveBeenCalledTimes(exp)</code>	Expect the function spied on to have been called an expected number of times
<code>toHaveBeenCalledWith(obj)</code>	Expect the function spied on to have been called with these particular arguments at least once

## Jasmine spyOn, createSpy and createSpyObj

`spyOn(obj, methodName)` returns a `Spy` object

- `obj` – `Object` – The object on which to install the `Spy`

- `methodName` – `String` – The name of the method to replace with a `Spy`

`jasmine.createSpy([name, originalFn])` returns a `Spy` object

- `name` – `String` – Optional – Name given to the spy, displayed in failure messages

- `originalFn` – `function` – Optional – Function to act as the real implementation

`jasmine.createSpyObj([baseName], methodNames)` returns an `Object`

- `baseName` – `String` – Optional – Base name for the spies in the object

- `methodNames` – `Array.<String>|<Object>` - Array of method names to create spies for OR `Object` who's keys will be method names and values the `returnValue`

## Example Spy Spec

```
// in SpyJasmine.js
const Dictionary = () => {};
Dictionary.prototype.hello = () => `hello`;
Dictionary.prototype.world = () => `world`;
const Person = () => {};
Person.prototype.sayHelloWorld = dictionary => {
  console.log(`${
    dictionary.hello()
  } ${dictionary.world()}`);
}

// in SpyJasmineSpec.js
describe(`Example of Jasmine Spy using spyOn()`, () => {
  it(`uses the dictionary to say "Hello World"`, () => {
    const dictionary = new Dictionary();
    const person = new Person();

    spyOn(dictionary, `hello`); // replace hello fn with spy
    spyOn(dictionary, `world`); // replaced world fn with another spy
    person.sayHelloWorld(dictionary);

    expect(dictionary.hello).toHaveBeenCalled();
    // not possible without first spy
    expect(dictionary.world).toHaveBeenCalled();
    // not possible without second spy
  });
});
```

## Example SPY Spec

```
// in SpyJasmineSpec.js
describe(`Example of Jasmine Spy using createSpy()`, () => {
  it(`uses the dictionary to say "Hello World"`, () => {
    const dictionary = new Dictionary();
    const person = new Person();
    dictionary.hello = jasmine.createSpy(`hello spy`);
      // replace hello fn with spy
    dictionary.world = jasmine.createSpy(`world spy`);
      // replaced world fn with another spy
    person.sayHelloWorld(dictionary);
    expect(dictionary.hello).toHaveBeenCalled();
    // not possible without first spy
    expect(dictionary.world).toHaveBeenCalled();
    // not possible without second spy
  });
});
```

## Example SPY Spec

```
// in SpyJasmineSpec.js
describe(`Example of Jasmine Spy using createSpyObj()`, () => {
  it(`uses the dictionary to say "Hello World"`, () => {
    const person = new Person();
    // Replace the dictionary with a mocked spy object
    const dictionary = jasmine.createSpyObj(dictionary, [hello, world]);
    person.sayHelloWorld(dictionary);
    expect(dictionary.hello).toHaveBeenCalled();
    // not possible without the spy [hello] inclusion
    expect(dictionary.world).toHaveBeenCalled();
    // not possible without the spy [world] inclusion
  });
});
```

## Set Up and Tear Down

Code can be executed before and after each spec or suite is run

Jasmine provides the following functions to facilitate this

- Each receives can receive a function to execute (and a timeout for asynchronous use):

BEFORE		AFTER	
<b>beforeEach()</b>	<b>beforeAll()</b>	<b>afterEach()</b>	<b>afterAll()</b>
Runs before each <code>it()</code> calls execute when present at the start of a <code>describe()</code> block	Runs once, before any of the <code>it()</code> calls are executed  <i>Seen as dangerous as can result in leaking state between specs producing erroneous passes/fails</i>	Runs straight after each of <code>it()</code> call executes when present	Runs after all of the <code>it()</code> calls have executed  <i>Seen as dangerous as can result in leaking state between specs producing erroneous passes/fails</i>

# Asynchronous Testing in Jasmine

Sometimes asynchronous functionality is tested

Specs and the setup and teardown methods can be made asynchronous by adding the `waitForAsync` keyword before their callbacks

```
beforeEach(waitForAsync() => await someAsyncSetUpFunction(); );
it(`does something async`, async() => {
  const someAsyncResult = await doSomethingAsync();
  ...
});
```

Matcher	Description
<code>toBeRejected()</code>	Expect a <code>Promise</code> to be rejected
<code>toBeRejectedWith(expected)</code>	Expect a <code>Promise</code> to be rejected with a value === to the expected
<code>toBeResolved()</code>	Expect a <code>Promise</code> to be resolved
<code>toBeResolvedTo(expected)</code>	Expect a <code>Promise</code> to be resolved to a value === to the expected

# Test Debugging

Karma browser allows debugging of tests

- Run a test as usual using the `npm test` command
- In the Karma browser, click on the `DEBUG` button to open a new Karma browser window with testing enabled
- Navigate to the `Sources` section and set a breakpoint in the test that you want to debug
- Refresh the browser window to re-run the tests  
→ The tests should pause at the breakpoint

The screenshot shows the Karma v3.1.4 browser interface. At the top, a green bar displays "Karma v3.1.4 - connected" and a "DEBUG" button. Below the bar, the status "Chrome 77.0.3865 (Mac OS X 10.14.6) is idle" is shown. The main area has tabs for Sources, Network, Performance, Memory, Application, Security, and Audits. The Sources tab is active, showing code from "simple.service.spec.ts". A breakpoint is set on line 10, which is highlighted in green. The code is as follows:

```
4 import { TestBedRender3 } from '@angular/core/testing/src/r3_test_bed';
5
6 describe('SimpleService', () => {
7   beforeEach(() => TestBed.configureTestingModule({}));
8
9   it('should be created', () => {
10     const service: SimpleService = TestBed.get(SimpleService);
11     expect(service).toBeTruthy();
12   });
13 });
14
15 describe('Testing simple service using just Jasmine', () => {
16   let service: SimpleService;
17   const expectedData = ['some', 'sample', 'data'];
18
19   beforeEach(() => { service = new SimpleService(); });
20
21 })
```

At the bottom of the code editor, it says "Line 1, Column 1" and "(source map Screenshot)".

# Code Coverage

Unit tests ran from the CLI can create code coverage reports

- Reports show parts of code base not properly tested by unit tests

Use command:

```
ng test --no-watch --code-coverage
```

- Results stored in **/coverage** folder
  - **index.html** will contain the report

Alter **angular.json** config file to run every time

```
"test": {  
  "options": {"codeCoverage": true }  
}
```

```
[10] building 5/6 modules 1 active ...Testing/solution/src/main/rn/spec.tsx [24 / 209 16:15:19.124]:INFO [Karma-server]: Karma v3.1.4 server started at http://0.0.0.0:9876  
[10] 2019-10-20T16:15:19.136 [launcher]: Launching browser Chrome with concurrency unlimited  
[10] building 5/6 modules 2 active ...Testing/solution/src/main/styles.cssBrowsersList: caniuse-lite is outdated  
[10] Please run next command npm update --save-dev caniuse-lite browserslist  
[10] building 7/6 modules 1 active ...Testing/solution/src/main/rn/spec.tsx [24 / 209 16:15:19.288]:INFO [launcher]:  
[10] 2019-10-20T16:15:27.783 [INFO] [Chrome_77.0.3865 (Mac OS X 10.14.6)]: Executed on socket MKIRj0sJ41lTBYCAA with i  
[10] 45193688  
[10] 0.388s ( Mac OS X 10.14.6 ): Executed 21 of 21 SUCCESS ( 0.371 secs / 0 .328 secs )  
TOTAL: 21 SUCCESS
```



## **QuickLab 3 – Jasmine**



- No activities



## Objectives

- To understand how a Jasmine Unit Test is written
- To be able to use spies in tests
- To be able to use functions to set up and tear down tests
- To understand how asynchronous testing can be done
- To be able to debug Jasmine tests
- To be able to generate code coverage reports from Jasmine tests



Angular Logo from: <https://angular.io/presskit>

Jasmine Logo from: <https://jasmine.github.io/>



# Components

Building web applications using angular



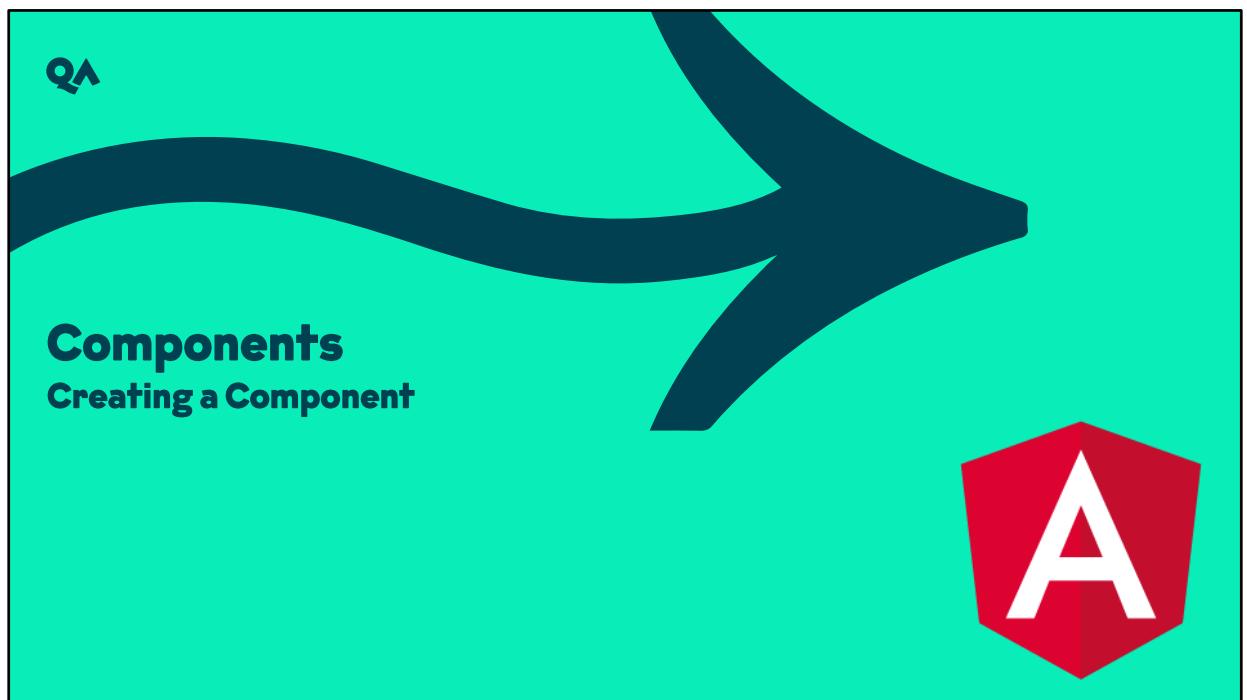


## Objectives

- To be able to create a component
- To be able to create and edit component templates and style
- To be able to bind data from:
  - Component to Template
  - Template to Component
  - 2-way Databinding
- To understand and be able to use Life-cycle hooks
- To know how to test Components
  - As a class
  - With the DOM



Angular Logo from: <https://angular.io/presskit>



Angular Logo from: <https://angular.io/presskit>

# Components

Components are the fundamental building blocks of the interface of Angular applications

Angular applications are a tree of components

A component is made up of:

- A **class** which contains the application logic
- A **template** which contains the instructions on how to construct the UI

A component must belong to an **NgModule**

The `@Component()` decorator declares a class to be a component and tells Angular how to construct, process and use at runtime

## Creating a new component

Creating a new component is as straightforward as declaring a class with an `@Component` decorator

The most commonly used configuration options used within the decorator:

- `selector` – the CSS selector that identifies this component in a template. Best practice: this should be an element
- `template/templateUrl` – the template for the view
- `styles/styleUrls` – inline CSS or links to external stylesheets applied to the view

Can be created using the command:

```
ng generate component <ComponentName>    OR    ng g component <ComponentName>
```

- Be sure to point the command line to the location the component should be created in
- Creates the TypeScript, HTML and CSS files for the component (along with a `.spec` test file)
- Automatically imports and declares the component in closest Module found upwards in the folder structure

## Adding a new component to the app render

Our newly created component can be made a child of other components

Use the **selector** text as an HTML style element in the template of any other component

- For example, a component with a selector of **my-new-component** could be added to the template of another component declared in the same module as shown:

```
<h1>My App Component</h1>
<p>
  Below me is where I will call for the MyNewComponent to be rendered
</p>

<my-new-component></my-new-component>

<p>
  Several components may be nested in a complex app
</p>
```

## **QuickLab 4a – Creating a new Component**



In this QuickLab you will:

- Create a new Component using the CLI
- Nest the component in the existing App component

QA

## Components

Creating a component template

Data Binding:

Component to View

View to Component

2-way



Angular Logo from: <https://angular.io/presskit>

# Templates

Made up of a mixture of HTML, CSS and other data from Angular bound in the form of:

- Interpolation
- Template Expressions
- Template Statements
- Event, Class, Attribute and other binding methods

Styling is applied to individual Components using the **styles** or **styleUrls** **metadata**

- Inline style can be provided by styles in the form of a **string** of CSS declarations
- Several stylesheets can be used when included in the **array** of **styleUrls**

A simple template can just contain HTML!

# Data Binding

Components act as the Controller/ViewModel, where as templates act as the View

Data binding provides a mechanism for coordinating what the user sees, with data in the application

Angular has multiple binding types depending on use:

- One-way (data source to view target)
- One-way (view target to data source)
- Two-way

# Data Binding

Data binding is a way to coordinate what users see using application values.

Simply declare bindings between binding sources and target HTML elements whilst Angular does the rest!

Data Direction	Syntax	Type
One-way from data source (Component) to view target (Template)	<code>{{expression}}</code> <code>[target]="expression"</code> <code>bind-target="expression"</code>	Interpolation Property Attribute Class Style
One-way from view target (Template) to data source (Component)	<code>(target)="statement"</code> <code>on-target="statement"</code>	Event
Two-way	<code>[(target)]="expression"</code> <code>bindon-target="expression"</code>	Two-way

## One-way data binding (data source to view target) – Interpolation

Interpolated `{{expressions}}` insert the result of the template expression at this point in the template

- These can be the evaluation of an expression as well as the value of a variable

```
// In Component class
public readonly text = `lorem ipsum dolot...`;

// In Component template
<p>{{text}}</p>
```

## One-way data binding (data source to view target) – Property, Class, Attribute and Style

Type	Target	Example
Property	Element, Component or Directive Property	<pre>&lt;img [src]="imageUrl"&gt; &lt;app-course [course]="currentCourse"&gt;&lt;app-course&gt; &lt;div [ngClass]="{{ 'special' : isSpecial }}&gt;</pre>
Class	<b>class</b> Property	<pre>&lt;div [class]="qaClasses"&gt;QA Ltd&lt;/div&gt; &lt;div [class.qaClasses]="true"&gt;QA Ltd&lt;/div&gt;</pre>
Attribute	Attribute	<pre>&lt;button [attr.aria-label]="help"&gt;Help&lt;/button&gt;</pre>
Styling	<b>style</b> Property	<pre>&lt;p [style.color]="isSpecial ? 'hotpink' : 'red'"&gt;Text&lt;/p&gt; &lt;p [style.fontSize.em]="selected ? 2 : 1"&gt;Text&lt;/p&gt;</pre>

Property bindings set the **DOM element's property** to the result of the expression  
(but only if it's not the name of a known property directive, as with ngClass)

Gotcha! If you omit the [square brackets] then Angular will not evaluate the expression and instead treats it as a string

```
 <!-- almost certainly a 404 -->
```

## One-way data binding (view target to data source) – Events

Not all users sit passively staring at our websites. Sometimes they like to interact. In these cases data needs to flow from the element to the component

```
<button (click)="onSubmit()" type="submit">Submit</button>
<!-- or canonical form-->
<button on-click="onSubmit()" type="submit">Submit</button>
```

Angular will check for a matching event property on a known directive first

```
<button (qaClick)="clickEvent=$event" type="submit">Submit</button>
```

When an event is raised, information about that event is stored in **\$event**, whose shape is dependent on the target event

- If the target is a DOM element then **\$event** is a native DOM element event

## Two-way binding (view target to data source)

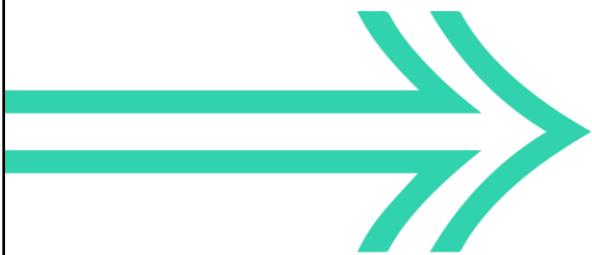
For when you want to display data from the component and update when the user changes

Combines the syntax of Property [] and Event () binding – giving us: Banana in a box [()] syntax

```
<input [(ngModel)]="username">
```

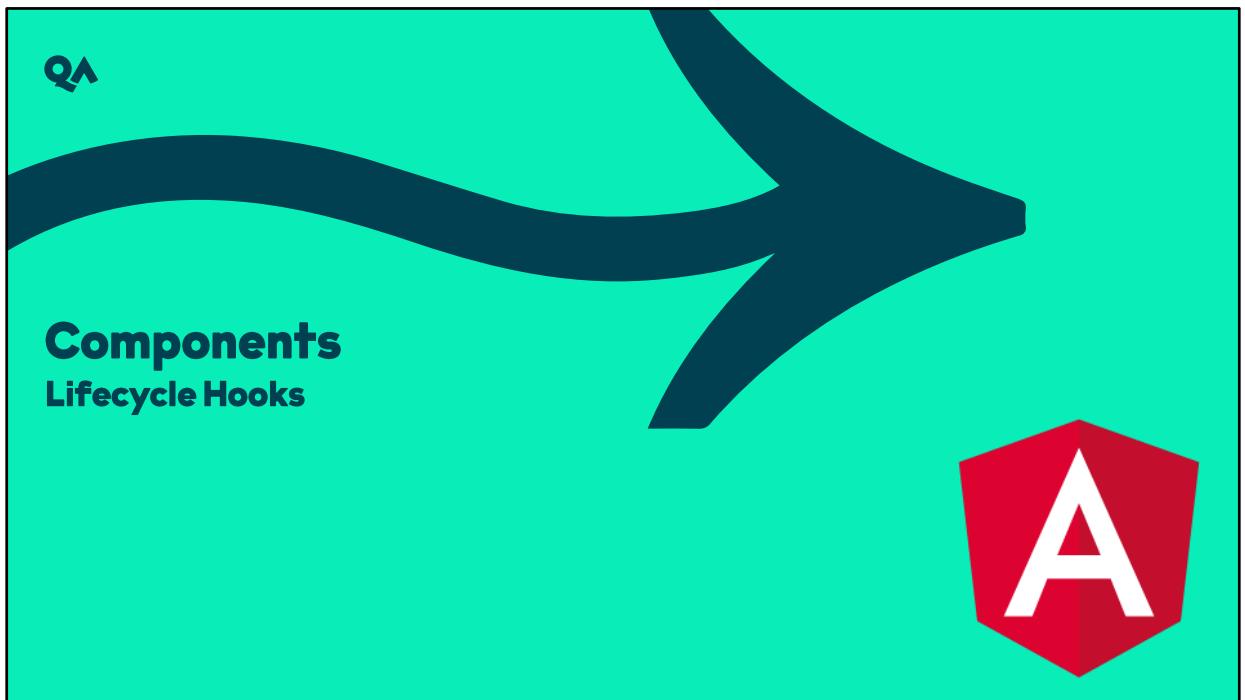
To use form elements with **ngModel**, the **FormsModule** must be imported into the **Module** the Component belongs to

## **QuickLab 4b – Data Binding**



In this QuickLab you will:

- Modify a component's template and styling
- Pass some data from the component to the template
- Pass some data from the template to the component
- Perform 2-way data-binding using NgModel



Angular Logo from: <https://angular.io/presskit>



## Lifecycle Hooks

Lifecycle Hook	Called	Other information
<code>ngOnChanges</code>	Whenever an input data-bound property is (re)set	First lifecycle hook to be called and the first time the data-bound properties are available to us
<code>ngOnInit</code>	Once following <code>ngOnChanges</code> after Angular first displays the data-bound properties and sets the component's input properties	Place any significant initialisation logic for the component in this method, leaving the constructor as simple as possible (ideally just property assignments)
<code>ngDoCheck</code>	During <b>every</b> change detection run, immediately following <code>ngOnChanges</code> (and <code>ngOnInit</code> if this is the first run)	Place logic here to detect any changes that Angular can't or won't detect on its own
<code>ngAfterContentInit</code>	Only ever once, following the first <code>ngDoCheck</code>	Place code here to respond to Angular projecting external content into the component's view
<code>ngAfterContentChecked</code>	After every <code>ngDoCheck</code> (after <code>ngAfterContentInit</code> if it is the first run)	Use this method to respond once Angular has checked the content projected into the component
<code>ngAfterViewInit</code>	Once after <code>ngAfterContentChecked</code>	Gives us opportunity to respond once Angular has initialised the component's view and child views
<code>ngAfterViewChecked</code>	After every <code>ngAfterContentChecked</code> (after <code>ngAfterViewInit</code> if the first run)	Called after Angular has checked the component's views and child views,
<code>ngOnDestroy</code>	Called just before Angular destroys the component	Perform any clean-up required to avoid memory leaks – common tasks will be unsubscribing from Observables and detaching event handlers



QA

## Component Testing

Testing the Class with no  
Dependencies

Testing the Class with  
Dependencies

Testing the DOM



Angular Logo from: <https://angular.io/presskit>

Jasmine Logo from: <https://jasmine.github.io/>

# Component Testing

Components are complex

- Combines HTML templates and TypeScript classes
- Need to test that these things work together properly

Component's class can be tested easily

- Test class functions on their own in as you would any TypeScript or JavaScript code

Testing the DOM interactions is trickier:

- Need to create component's host element in the Browser DOM and make sure component's class interacts with template as expected

**TestBed** enables these tests to be done

## Component Class Testing (No dependencies)

Sufficient to check that the click of the button changes the state in the class

```
@Component({
  selector: `app-root`,
  template: `
    <button (click)="toggleState()">
      Toggle
    </button><span>{{message}}</span>`
})
export class AppComponent {
  flag = false;
  toggleState() {
    this.flag = !this.flag;
  }
  get message() {
    return `The status is ${this.flag}`;
  }
}
```

Consider a component that would toggle the show state of a message

```
...
describe(`Button tests`, () => {
  it(`#toggleState should toggle #flag`, () => {
    const component =
      new AppComponent ();
    expect(component.flag).
      toBe(false, `false at first`);

    component.toggleState();
    expect(component.flag).
      toBe(true, `true after click`);

  });
...
})
```

## Testing Components with the DOM

Class tests only tell you about the class behaviour

Unknown if component will:

- Render properly
- Respond to user input/control
- Integrate with parent and child components correctly

Complex component interactions with the DOM occur – through structural directives, etc.

- Need DOM elements associated with components

To be able to confirm that the component has rendered correctly at particular times

So that user interaction can be simulated to check that the interactions produce expected behaviour

Additional features of **TestBed** and other testing helpers will be used

## Testing Components with the DOM

```
import { TestBed } from '@angular/core/testing';
import { AppComponent } from './app.component';
describe('AppComponent - minimal with beforeEach', () => {
  let component: AppComponent;
  let fixture: ComponentFixture<AppComponent>
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [AppComponent]
    });
    fixture =
      TestBed.createComponent(AppComponent);
    component = fixture.componentInstance;
  });
  it(`should create`, () => {
    expect(component).toBeDefined();
  });
});
```

Specs automatically created when using CLI to generate app and new components

Boilerplate provided for substantial testing of components

- Can be reduced to simplify understanding

# Testing Components with the DOM

```
import { TestBed } from '@angular/core/testing';
import { AppComponent } from './app.component';
describe('AppComponent - minimal with beforeEach)', () =>
{
  let component: AppComponent;
  let fixture: ComponentFixture<AppComponent>
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [AppComponent]
  });
  fixture =
    TestBed.createComponent(AppComponent);
  component = fixture.componentInstance;
});
it(`should create`, () => {
  expect(component).toBeDefined();
});
});
```

Metadata object passed to `TestBed.configureTestingModule` defines the component to test

fixture defined using `TestBed's createComponent` method

- Creates an instance of the component and adds element to test-runner DOM, returning a `ComponentFixture`

→ Test harness for interacting with created component and its element

# Testing Components with the DOM

```
...
it(`should contain "it works!"`, () => {
  const myElement: HTMLElement =
    fixture.nativeElement;
  expect(myElement.textContent).toContain(`it works!`);
});
it(`should have a <p> with `text`, () => {
  const myElement: HTMLElement =
    fixture.nativeElement;
  const p = myElement.querySelector('p');
  expect(p.textContent).toEqual(`text`);
});
...

```

`ComponentFixture.nativeElement` allows inspection of the DOM element

- Properties of this can be examined
  - Eg `textContent` as shown
- Has type `any`
  - Not known what type of element it will be if one at all!
  - Tests designed to run in browser so `nativeElement` is always a derivative of `HTMLElement`
- Can use `querySelector` to examine its tree

84

# Testing Components with the DOM

```
import { DebugElement } from '@angular/core';
...
it(`should have a <p> with `text`, () => {
  const myElementDe: DebugElement =
    fixture.debugElement;
  const myElementEl: HTMLElement =
    myElementDe.nativeElement;
  const p =
    myElementEl.querySelector('p');
  expect(p.textContent).toEqual(`text`);
});
...
...
```

`nativeElement` provides an `HTMLElement`

- Actually implemented as `fixture.debugElement.nativeElement`
- `nativeElement` properties depend on the runtime environment
- Could be running tests outside of a browser environment without DOM or DOM capabilities
- `DebugElement` abstraction allows testing to be done across all supported platforms
- `DebugElement` tree wraps `nativeElement` and then `nativeElement` unwraps it to return platform specific element

# Testing Components with the DOM

```
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-
browser';
...
it(`should have a <p> with `text`, () => {
  const myElementDe: DebugElement =
    fixture.debugElement;
  const paraDe =
    myElementDe(By.css('p'));
  const p: HTMLElement =
    paraDe.nativeElement;
  expect(p.textContent).toEqual(`text`);
});
...

```

**DebugElement** has own query methods

- Allows support for runtime test environments that do not support full **HTMLElement** API
- Takes a predicate function that returns true when node in **DebugElement** tree matches selection criteria
  - Predicate created with help of **By**
  - Imported from **platform-browser**
  - **css()** function accepts any valid CSS selector
  - Need to unwrap element found
  - Often easier and clearer to filter with **querySelector()** and **querySelectorAll()**

## Testing the DOM and bound data: Change Detection

```
...
it(`should have a p containing the message
when ngOnInit is called`, () => {
  component.ngOnInit();
  fixture.detectChanges();
  p = helloWorldEl.querySelector('p');
  expect(p).toBeTruthy();
  expect(p.textContent).
   toContain(
      helloWorldService.getMessage()
    );
});
...

```

Angular does not automatically bind data from the class to the template when `createComponent` is called

- Testing content from the DOM will result in a failed test

Binding occurs when Angular performs change detection

- Happens automatically in production
- Need to call `fixture.detectChanges()` in testing
  - Allows inspection at various points
  - Allows data to be changed in test

# Testing the DOM and bound data: Change Detection

```
...
@Component({
  selector: `app-nested-component`,
  template: ``
})
class NestedStubComponent { }

describe(`Test Suite`, ()=> {
  beforeEach(waitForAsync() => {
    TestBed.configureTestingModule({
      declarations: [
        CompUnderTest,
        NestedStubComponent
      ],
      .compileComponents();
    });
    ...
  });
});
```

To maintain the unit nature of components a strategy needs to be adopted to deal with nested component calls when testing

- Create a class within the Component's spec file that provides a stub for any component whose selector appears in the component-under-test's template
- Stub Component should have the selector used for the real component and an empty template
- Any required data, such as inputs can also be included in the Stub
- This Stub Component needs to be declared in the **TestBed** configuration along with the component-under-test

## **QuickLab 4c – Testing Class and DOM**



In this QuickLab you will:

- Write some Jasmine tests to ensure a component's class works as expected
- Examine the DOM element created by a component and check the expected data-binding has occurred



## Objectives

- To be able to create a component
- To be able to create and edit component templates and style
- To be able to bind data from:
  - Component to Template
  - Template to Component
  - 2-way Databinding
- To understand and be able to use Life-cycle hooks
- To know how to test Components
  - As a class
  - With

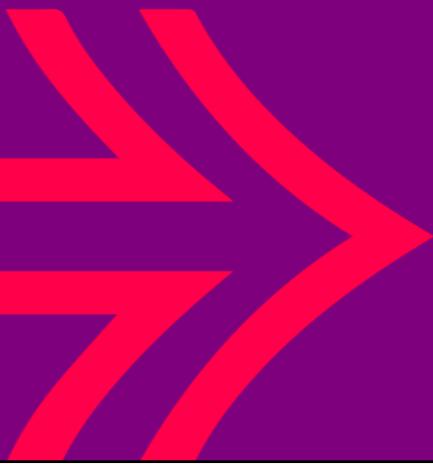


Angular Logo from: <https://angular.io/presskit>



## Directives

Building Web Applications using Angular





## Objectives

- To be able to use built-in directives
- To be able to use and test Input and Output properties
- To be able to create and test Custom Directives



Angular Logo from: <https://angular.io/presskit>



## Directives

### Built-In Directives

**NgClass**

**NgStyle**

**NgIf**

**NgFor**

**NgSwitch**



Angular Logo from: <https://angular.io/presskit>

# Introduction

Directives come in three forms in Angular

- Components – simply a directive with a template
- Attribute directives – changing the appearance or behaviour of an element, component or other directive
- Structural directives – changing the DOM through the addition or removal of elements

Angular provides built-in directives for commonly required features

Angular allows creation of custom directives to give developers freedom to implement directives in bespoke ways

## Built-in Directives

Angular comes with built-in directives to help us build our own components

Angular has attribute directives

- NgClass**
- NgStyle**

And structural directives

- NgIf**
- NgFor**
- NgSwitch**

## NgClass – built-in attribute directive

**NgClass** provides a mechanism for us to toggle multiple classes based on our data source

We need to point **NgClass** at an object which holds a series of **class:boolean** entries, these define what class names should be applied, if their Boolean value is **true**

We can change these values and see the classes added/removed

```
currentClasses = {  
  selected: false,  
  valid: false  
};  
  
<section [ngClass]="currentClasses">  
<button (click)="currentClasses.valid=true">Validate</button>
```

## NgStyle – built-in attribute directive

NgStyle works similarly to NgClass, with a `style:value` control object

With NgStyle the value of each style should resolve to an appropriate value for that style (not true or false as with NgClass)

We then map the control object to the NgStyle directive in the template

```
currentStyles = {}

setCurrentStyles() {
  this.currentStyles = {
    backgroundColor: this.selected ? 'lightgreen' : 'red',
    border: this.valid ? '2px solid green' : '2px solid lightpink'
  }
};

<section [ngStyle]="currentStyles">
<button (click)="valid=!valid; setCurrentStyles()">Validate</button>
```

## **NgIf – built-in structural directive**

**NgIf** adds elements from the DOM if the expression returns a truthy value  
And removes them if the expression returns falsey

```
<article *ngIf="viewArticle">...</article>
<button (click)="viewArticle=!viewArticle">Toggle Article</button>
```

If you just want to show/hide something then styles may be more suitable – this will keep the element in memory and Angular may continue to check for changes.

The \*ngIf would actually be as follows without the 'syntactic sugar' Angular provides:

```
<ng-template [ngIf]="viewArticle">
  <article>...</article>
</ng-template>
<button (click)="viewArticle=!viewArticle">Toggle Article</button>
```

## NgFor – built-in structural directive

NgFor allows us to build repetitive areas of the template based on the contents of collections

```
<ul>
  <li *ngFor="let user of users">{{user.username}}</li>
</ul>
```

The syntax “**let user of users**” is called microsyntax and is not a template expression  
You can also include:

- **index**
- **trackby**

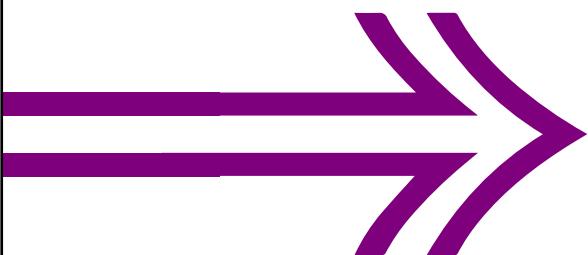
## **NgSwitch – built-in structural directive**

**NgSwitch** is actually a collection of directives that act together to create something analogous to the JavaScript switch statement

We use it to display an element based on a switch condition

```
<div [ngSwitch]="instructor">
  <p *ngSwitchCase="'chris'">Chris enjoys cycling!</p>
  <p *ngSwitchCase="'ed'">Ed just loves JavaScript!</p>
  <p *ngSwitchCase="'edsel'">Edsel loves a good group selfie</p>
  <p *ngSwitchCase="'dave'">
    Dave puts Star Wars references wherever he can
  </p>
  <p *ngSwitchDefault>Please select an instructor</p>
</div>
```

## **QuickLab 5a – Built-in Directives**



- In this QuickLab you will:
- Use attribute directives to affect the template
- Use structural directives to add multiple and conditional elements to the template



## Directives

**Input Properties**

**Output Properties**



Angular Logo from: <https://angular.io/presskit>

## Input Properties

Input properties allow the binding of a component property to a value

- Either hard coded or referencing a parent component's property value

Input properties are enabled through the use of the `@Input()` decorator which tells Angular that this property will be the target of a binding

- If you don't use this decorator you will get a template parse error should you try to bind to it

```
export class UserComponent {  
  @Input()  
  user: string;  
}
```

We create the binding in the same way we did earlier in this chapter, this time we are binding to our own defined property

```
<app-user [user]="instructor"></app-user>
```

## Input Property Alias

Sometimes we may wish to expose a different name to the one we use internally in the class

- Done by aliasing the input property by passing a string into the decorator

```
export class UserComponent {  
  @Input('user')  
  person: string;  
}
```

We can now use 'person' in the class, whilst in the template we continue to use 'user'

```
<app-user [user]="instructor"></app-user>
```

Note – this practice is discouraged, especially by the linter!

# Output Properties

Output properties allow us to inform a parent when a change within the child component has taken place

Initially we set up an Output property similarly to an Input property, using an `@Output()` decorator

- Need this property to be an instance of `EventEmitter`, imported from `@angular/core`

```
export class UserComponent {  
  @Output()  
  vote = new EventEmitter<number>();  
}
```

We then bind to this output property in the parent component

```
<app-user (vote)="handleVote($event)"></app-user>
```

## Output Properties

Our parent component then needs the `handleVote` method which receives the `$event`

```
handleVote(event) {  
    alert(`A vote has been received: ${event}`); //A vote has been  
    received: 1  
}
```

Our child component then simply emits events when it wants to notify the parent of a change

```
upvote(person) {  
    this.vote.emit(1);  
}  
  
downvote(person) {  
    this.vote.emit(-1);  
}
```

## Output Property Alias

Output properties may have aliases just as Input properties do, added (and discouraged ) in the same way:

```
export class UserComponent {  
    @Output('change')  
    vote = new EventEmitter<number>();  
}
```

We then bind to `change` (for the template) but use `vote` in the class as before

```
upvote(person) {  
    this.vote.emit(1);  
}  
downvote(person) {  
    this.vote.emit(-1);  
}
```

```
<app-user (change)="handleVote($event)"></app-user>
```

QA

## **Testing Input and Output Properties**

**Testing Inputs Properties**

**Testing Output Properties**



Angular Logo from: <https://angular.io/presskit>

Jasmine Logo from: <https://jasmine.github.io/>

## Testing @Inputs

As simple as testing a component class property

- Run tests as required to ensure that their value can change and any side-effects occur

## Testing @Outputs

Outputs usually emit some data – this will generally be Observable!

Need to subscribe to the value that is emitted

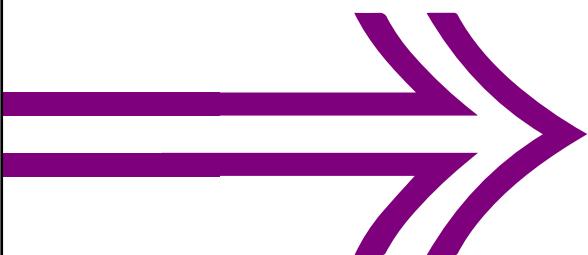
Can then check to see if this is what is expected

- Checks can be linked to UI events, like mouse clicks

`triggerEventHandler` can be called on the element

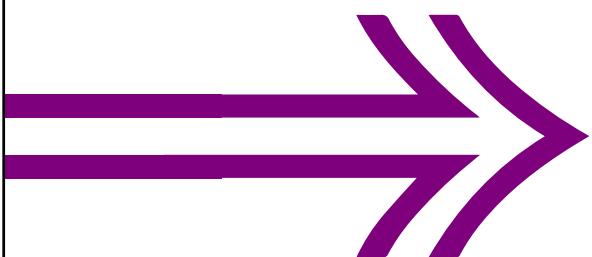
Pass the type of event to trigger (eg. Click)

## **QuickLab 5b – Input and Output Properties**



- In this QuickLab you will:
- Add an Input property to a component so that it can receive data from its parent
- Use an Output property to emit data to be caught by its parent

## **Quick Lab 5c – More Component Testing**



In this QuickLab you will:

- Stub nested components to allow proper unit testing of a parent
- Check that Input and Output functionality passes data as expected through tests



## Directives

### Custom Directives

**Creating**

**Reacting to Events**

**Taking Inputs**

**Structural Directives**



Angular Logo from: <https://angular.io/presskit>

## Custom Attribute Directives - Creation

Attribute directives make changes to an element and are simply classes with the Directive decorator added to them

```
@Directive({
    selector: '[appSpecial]'
})
export class SpecialDirective { }
```

The constructor of such a class is passed as `ElementRef` by Angular, which provides access to the underlying DOM element

```
export class SpecialDirective {
    constructor(private el: ElementRef) {
        el.nativeElement.style.fontSize = "5em"
    }
}

<p appSpecial>Some very large text indeed</p>
```

## Custom Attribute Directives – Reacting to Events

We can also listen for events on the element through the use of the `HostListener` decorator which we attach to methods to handle such events.

```
export class SpecialDirective {  
    ...  
  
    @HostListener('mouseenter') onMouseEnter() {  
        this.show();  
    }  
}
```

The method can have any name, it is the string passed to the decorator that describes the event we're listening for

## Custom Attribute Directives – Taking Input

Attribute Directives can take values just as we saw with Components

```
export class SpecialDirective {  
    @Input() fontSize: string|number  
}
```

Perhaps to then tailor the directive according to the value passed (remember our life cycle events?)

```
export class SpecialDirective {  
    @Input() fontSize: string|number  
  
    ngOnInit(private el: ElementRef) {  
        el.nativeElement.style.fontSize = this.fontSize  
    }  
}
```

```
<p appSpecial fontSize="5em">
```

## Custom Structural Directives

Structural directives are used to modify the DOM, they can be fairly simple like the built in `*ngIf` directive or quite complex like the built in `*ngFor` directive

They are created through the use of the `@Directive` to decorate applied to a class, it is then injected with a `TemplateRef` and `ViewContainerRef`

```
@Directive({
    selector: '[appLoggedIn]'
})
export class LoggedInDirective {
    constructor(
        private templateRef: TemplateRef<any>,
        private viewContainer: ViewContainerRef,
    ) { }
```

## Structural Directives – \*Syntax

The **\*syntax** we're used to seeing on Angular's structural directives is there to simplify our lives

- important to know what the expanded syntax looks like so that we understand why we build Structural Directives the way we do

```
//easy on the eyes syntax
<div *ngIf="bool">...</div>

//the same as writing
<ng-template [ngIf]="bool">
    <div>...</div>
</ng-template>
```

Angular parses the **\*syntax** into the **ng-template** version you see above.

- Take the directive up to an **ng-template** which then wraps the target element and its children

## Structural Directives – \*Syntax

```
//easy(ish) on the eyes syntax
<div *ngFor="let user of users; let i=index"
 [class.odd]="odd">
  {{i}} {{user.name}}
</div>

//the same as writing
<ng-template ngFor let-user [ngForOf]="users" let-
i="index">
  <div [class.odd]="odd">{{i}}
  {{user.name}}</div>
</ng-template>
```

- We can go further and look at **\*ngFor**
- The **let** keyword creates a template input variables (in this case, **user** and **i**)
- The microsyntax parser takes the **of** keyword, title-cases them and prefixes them with the directive name (**ngForOf**)
- The directive sets and resets context object properties, for **\*ngFor** these include **\$implicit** and **index**
- Our template input variable **i** gets assigned to the context object property **index** and as **user** did not specify a property, it receives **\$implicit**

119

## Structural Directives

Now that we know the syntax and the way it is parsed under the covers, we can write our own directives to take advantage of these features. Reminding ourselves of the basic construct:

```
@Directive({
    selector: '[appLoggedIn]'
})
export class LoggedInDirective {
    constructor(
        private templateRef: TemplateRef<any>,
        private viewContainer: ViewContainerRef,
    ) { }
```

The **templateRef** gives us access to the **<ng-template>** that angular creates for us when using the **\*syntax**

The **viewContainer** gives us the ability to render the view based on our logic and any data being passed back and forth

## Structural Directives

Angular creates a view-container adjacent to the host element (the one we placed the directive on)

This view-container is where we place an embedded view, created using the angular generated ng-template

```
this.viewContainer.createEmbeddedView(this.templateRef);
```

We can use some logic to intercept the creation of this view, such as how **\*ngIf** creates/destroys the view and **\*ngFor** creates many copies of the view.

```
if (bool) {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
}
```

```
for (...) {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
}
```

## Structural Directives - @Input variables

To pass data into the directive we use our input variables as we have done with Components and Attribute directives

```
<div *qaSpecial="inputData">...</div>
```

We use a setter here so that if `inputData` changes our logic is re-evaluated. No getter is required as no one is fetching this data.

```
@Input() set qaSpecial(inputData) {  
    //do something with inputData  
}
```

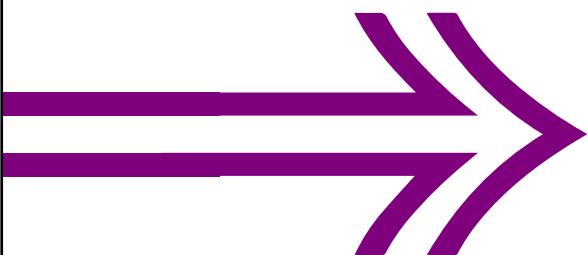
## Structural Directives – Template Input Variables

If we wish to use data returned from the directive in the rendering of our view, much like we saw with \*ngFor, then we need to use the context object

```
<div *qaSpecial="inputData; let text; let author=user">
    <h3>{{author}}</h3>
    <p>{{text}}</p>
</div>
```

```
if (bool) {
    this.viewContainer.createEmbeddedView(this.templateRef, {
        $implicit: `Here's some text for the unknown text variable`,
        user: `John Smith`           ←
    });
}
```

## **QuickLab 5d – Custom Directives**



- In this QuickLab you will:
- Create custom directive to be applied to a set of images
- Mouseover events will be monitored to change an image from grey-scaled to full colour



## **Testing Custom Directives**

**Setting up the tests**

**Test Wrapper Components**

**Interacting with and Inspecting the View**



Angular Logo from: <https://angular.io/presskit>

Jasmine Logo from: <https://jasmine.github.io/>

## Test Set Up

```
import { TestBed } from '@angular/core/testing';
import { TestDirective } from './test.directive';
describe(`TestDirective tests`, () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [TestDirective]
    });
  });
});
```

Directives often affect the display of an element on the screen

- Attribute directives are attached to elements and then applied
- Commonly user `@HostListener` trigger the directive to be applied dependent on a particular event, supplying a particular value

Need a test suite for the Directive that configures the testing module, declaring the Directive-Under-Test

# Test Wrapper Components

```
import { TestBed } from '@angular/core/testing';
import { TestDirective } from './test.directive';
import { Component } from '@angular/core';
@Component({
  template: ``
})
class TestComponent
describe(`TestDirective tests`, () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [TestDirective,
        TestComponent]
    });
  });
});
```

Directive usually performs some action on a template

- Need a simple Component with a template that will allow directive to be applied
- Can then apply tests to the Component to ensure that the Directive is applied correctly

Simple component can be a Class in the Directive's Spec

- It needs to be declared in the **TestBed**

## Interacting and Inspecting the View

```
...
it(`should change the border color when
clicked`, () => {
  imageEl.triggerEventHandler(`click`,
null);
  fixture.detectChanges();

expect(imageEl.nativeElement.style.border).
  toBe(`solid 5px blue`);
});

...

```

`TestComponent` can be rendered and inspected in the usual way

`triggerEventHandler()` can be used to excite the required element

- `fixture.detectChanges()` must be run after each

`nativeElement.style` can be used to access properties set by the Directive and check the expected outcomes



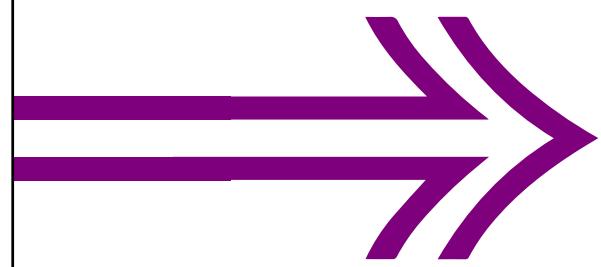
## Objectives

- To be able to use built-in directives
- To be able to use and test Input and Output properties
- To be able to create and test Custom Directives



Angular Logo from: <https://angular.io/presskit>

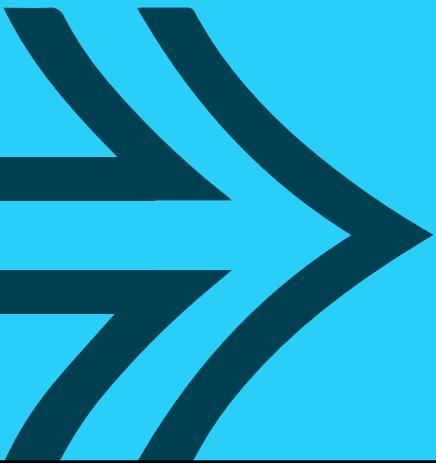
## **QuickLab 5e – Testing Custom Directives**

- 
- In this QuickLab you will:
  - Write Jasmine tests to ensure that the custom greyscale directive behaves as expected



# Observables

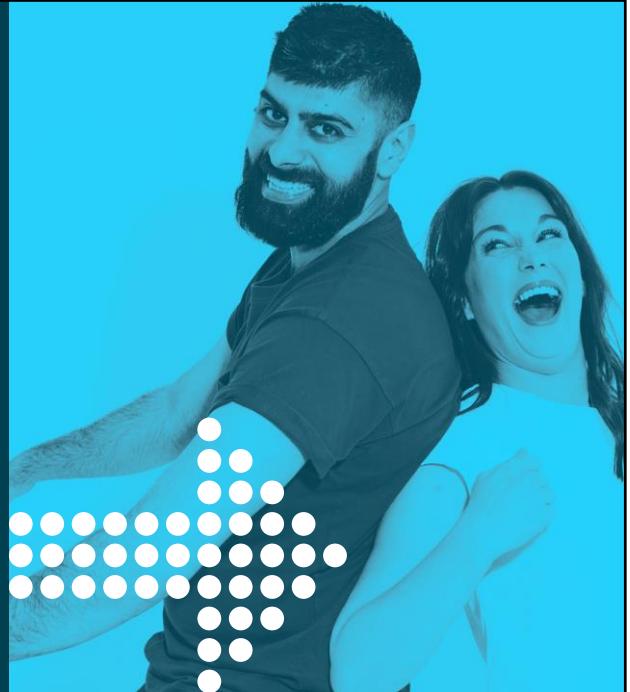
Building Web Applications using Angular





## Objectives

- To understand what an Observable is
- To understand what RxJS is
- To be able to use RxJS operators to work with Observables
- To understand how Observables can be used in Angular



Angular Logo from: <https://angular.io/presskit>



## Observables

**Observable Anatomy**

**Creating an Observable**

**Creating an Observer**

**Subscribing to an Observable**

**Handling Errors**



Angular Logo from: <https://angular.io/presskit>

# Introduction

Applications often need to handle events, asynchronous data delivery and multiple values  
Observables offer support for passing messages between publishers and subscribers in the application

- Declarative – define functions for publishing values not executed until consumer subscribes to it

Subscribed consumer is notified until function completes or they unsubscribe

- Can deliver multiple values of any type dependent on context

API same for synchronous and asynchronous data

Application only need worry about consuming values and unsubscribing

Observables handle setup and teardown logic

Used extensively within Angular

- But not limited to use within – they are pure JavaScript!

## Observable Anatomy

Publishers:

- Create an Observable instance
- Define a **subscriber** function

Executed when consumer calls `subscribe()` method

Defines how to obtain or generate values or messages to be published

Executing:

- Call the `subscribe()` method passing an observer JS object that defines handlers for notifications

Defines a **Subscription** object

Call `unsubscribe()` method on it to stop receiving notifications

## Creating an Observable

Is fairly simple to create an Observable

```
const myObservable = new Observable ((observer) => {
  const {next, error} = observer;
  let observableMessage;
  observableMessage = setInterval(() => {
    next('hello world!');
  },1000);
});
```



## Subscribing to an Observable

Observable instance begins publishing values only when at least 1 subscription is made to it

- To subscribe to an observable

```
let sub:Subscription = myObservable.subscribe(  
    next => console.log(next),           // onNext()  
    error => console.error(error),       // onError()  
    ()=>console.log('All done')         // onComplete()  
) ;
```

We have to be careful to unsubscribe from the Observable to not create memory leaks

```
setTimeout(()=>{  
    console.log("unsubscribing");  
    sub.unsubscribe();  
,10000)
```

## Handling Errors

Try/catch does not effectively catch errors as values are produced asynchronously

Errors handled by the error callback on the observer

Producing error causes the observable to:

- Clean up subscriptions
- Stop producing values (i.e. complete)



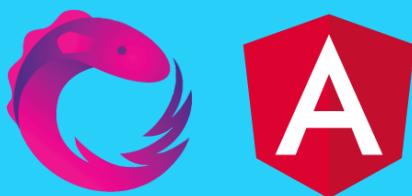
## RxJS

**What is RxJS?**

**Async management with RxJS  
Operators**

**The pipe function**

**Creation Operators**



Angular Logo from: <https://angular.io/presskit>

# RxJS

Reactive Extensions for JavaScript

- Library for reactive programming using observables

Makes it easier to compose asynchronous or callback-based code

- Reactive programming – asynchronous programming paradigm
- Provides implementation of Observable
- Provides utility functions for creating and working with observables

Converting existing code for async operations into observables (events, Promises, fetch, etc.)

- Iterating through values in a stream
- Mapping values to different types
- Filtering streams
- Composing multiple streams

# Essential concepts in RxJS for Async management

Concept	Description
<b>Observable</b>	Represents idea of an invokable collection of future values or events
<b>Observer</b>	A collection of callbacks that knows how to listen to values delivered by Observable
<b>Subscription</b>	Represents execution of an Observable – useful for cancelling execution
<b>Operators</b>	Pure functions that enable functional programming style of dealing with collections eg. map, filter, concat, flatMap, etc.
<b>Subject</b>	Equivalent to an EventEmitter - only way of multicasting a value or event to multiple Observers
<b>Schedulers</b>	Centralised dispatchers to control concurrency – allows coordination when computation happens on setTimeout, requestAnimationFrame, etc.

## RxJS Operators

Functions that build on Observables to enable more sophisticated manipulation of data:

- Eg. `map()`, `filter()`, `concat()`, `flatMap()`
- Takes configuration options
- Returns a function that takes a source observable

When executing the return function, operators can:

- Observe source observable's emitted values
- Transform the emitted values and return a new observable of transformed values

For more on RxJS operators see: <https://www.learnrxjs.io/operators/>

# Types of RxJS Operators

There are several different flavours of operators:

Operator Type	Description	Examples
Combination	Allow joining of information from multiple observables	<code>concat, concatAll, combineAll, combineLatest, merge, mergeAll</code>
Conditional	For use-cases when specific condition must be met	<code>defaultIfEmpty, every, iif, sequenceEqual</code>
Creation	Allow creation of observable from nearly anything – turn everything into a stream	<code>create, from, fromEvent, interval, of, timer</code>
Error Handling	Gracefully handle errors and retry logic should they occur	<code>catch/catchError, retry, retryWhen</code>

For more on RxJS operators see: <https://www.learnrxjs.io/operators/>

# Types of RxJS Operators

There are several different flavours of operators:

Operator Type	Description	Examples
Multicasting	Allows side-effects to be shared among multiple subscribers	<code>publish, multicast, share, shareReplay</code>
Filtering	Provide techniques for accepting particular values from an observable	<code>audit, filter, first, last, sample, skipWhile</code>
Transformation	Transform values as they pass through operator chain	<code>map, concatMap, mergeMap/flatMap, switchMap</code>
Utility	Useful utility functions that allow operations like logging, handling notifications, setting up schedulers, etc.	<code>do/tap, delay, repeat,toPromise</code>

For more on RxJS operators see: <https://www.learnrxjs.io/operators/>

## The pipe() function

```
import { Observable } from 'rxjs';
import { map, filter, scan } from 'rxjs/operators';

const data = interval(1000)
  .pipe(
    filter(x => x % 2 === 0),
    map(x => x + x),
    scan(acc, x) => acc + x);
let sub = data.subscribe(x =>
  console.log(x));
```

The pipe function can be used to stitch together functional operators into a chain

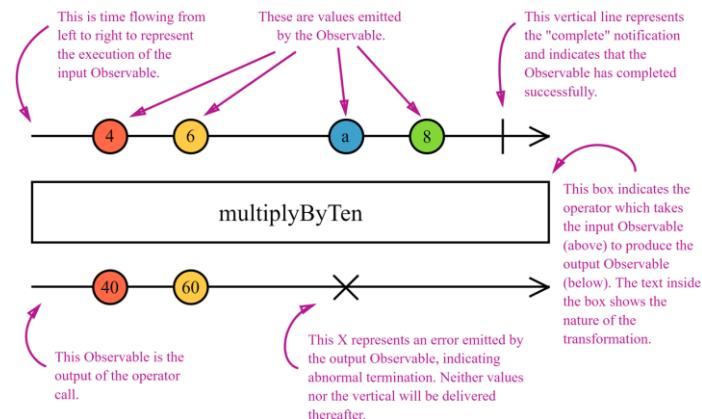
It returns an Observable

- Observable is result of all operations having been called in order they were passed in

# Marble Diagrams

Marble diagrams provide a visual queue as to how operators work

Given many operators are in some way related to time, text descriptions are often insufficient and confusing



All marble diagrams featured are sourced from ReactiveX© <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

# Observable Creation Operators

RxJS functions that can be used to create Observables

- Simplify process of creating observables from events, timers, promises, etc.
- They include:

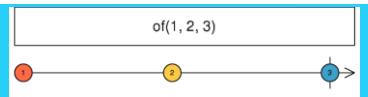
Creation Function	Purpose
<code>of</code>	Create an observable that emits a variable amount of values in sequence
<code>from</code>	Create an observable FROM a promise, etc.
<code>interval</code>	Create an observable that publishes a value on an interval
<code>fromEvent</code>	Create an observable that publishes when an event is raised – eg. mouse movement
<code>timer</code>	Create an observable that publishes a value after a given delay and then at another given delay (if supplied)
<code>ajax</code>	Create an observable that will create an AJAX request

For more on RxJS Creation Operators, see:

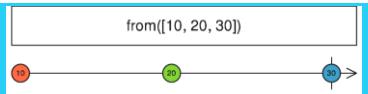
<https://www.learnrxjs.io/operators/creation/>

## Observable Creation Functions

```
import { of } from 'rxjs';
// Create an Observable out of emitted values
const data = of(1, 2, 3);
```



```
import { from } from 'rxjs';
// Create an Observable out of a some data
const data = from([10, 20, 30]);
```



```
import { interval } from 'rxjs';
// Create an Observable that publishes a value
// on an interval
const data = interval(1000);
```



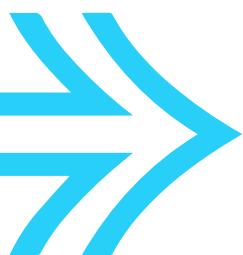
```
import { fromEvent } from 'rxjs';
// Create an Observable that publishes mouse movements
const el = document.querySelector(`#my-element`);
const data = fromEvent(el, `mousemove`);
```

```
import { ajax } from 'rxjs';
// Create an Observable that creates an AJAX request
const data = ajax(`/api/endpoint`);
```

All marble diagrams featured are sourced from ReactiveX® <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

For more on RxJS Creation Operators, see:  
<https://www.learnrxjs.io/operators/creation/>

## QuickLab 6a – Part 1 - Creation Operators



In this QuickLab you will:

- Use the `of` and `fromEvent` creation operators to create an **Observables** and subscribe to them

QA

# RxJS

## Operators

### Filtering Operators



Angular Logo from: <https://angular.io/presskit>

## Observable Filtering Operators

RxJS functions that can be used to select particular values from Observables

- They include:

Transform Function	Purpose
<code>filter</code>	Emit values that pass the provided condition
<code>take</code>	Emit the provided number of values before completing
<code>distinctUntilChanged</code>	Only emit when the current value is different than the previous value
<code>takeUntil</code>	Take values until another observable emits

- For RxJS v6+ these should be imported individually from the `rxjs/operators` package

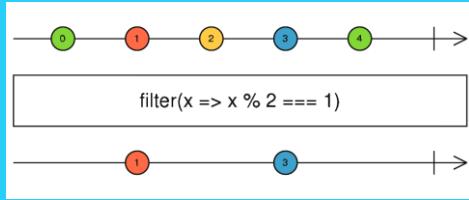
For more on RxJS Transformation Operators, see:

<https://www.learnrxjs.io/operators/filtering/>

# RxJS filter Operator

```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';
// Create observable using of that emits 0-4
const data1 = of(0, 1, 2, 3, 4);
// Emit values from data1, filtering out even values
const example = data1.pipe(filter(x => x % 2 ===
1));
// Subscribe to the observable and log the values
const sub = example.subscribe(x => console.log(x));

//Output:
// 1
// 3
```



Simple example of the **filter** operator

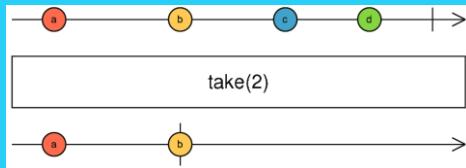
All marble diagrams featured are sourced from ReactiveX® <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

# RxJS take Operator

```
import { of } from 'rxjs';
import { take } from 'rxjs/operators';
// Create observable using of that emits 0-4
const data1 = of(`a`, `b`, `c`, `d`);
// Emit only the first 2 values from data1

const example = data1.pipe(take(2));
// Subscribe to the observable and log the values
const sub = example.subscribe(x => console.log(x));

//Output:
// a
// b
```



Simple example of the  
**take** operator

All marble diagrams featured are sourced from ReactiveX® <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

## QuickLab 6a – Part 2 - Filter Operators



In this QuickLab you will:

- Use the `of`, `filter` and `take` operators to create **Observables** and filter their output

QA

## RxJS Operators Combination Operators



Angular Logo from: <https://angular.io/presskit>

## Observable Combination Operators

RxJS functions that can be used to allow the joining of values from multiple Observables

- Order, time and structure of emitted values is main difference in the operators provided
- They include:

Transform Function	Purpose
<code>concat</code>	*Subscribe to observables in order as previous completes, emit values
<code>merge</code>	*Turn multiple observables into a single observable
<code>startWith</code>	**Emit given value first – value not given as part of observable
<code>withLatestFrom</code>	**Also provide last value from another observable

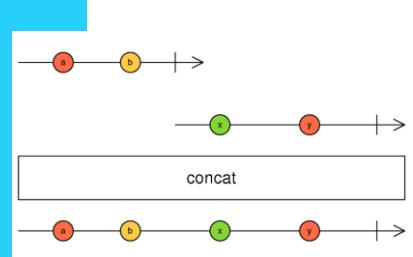
- \*For RxJS v6+ these should be imported individually from the `rxjs` package
- \*\*For RxJS v6+ these should be imported individually from the `rxjs/operators` package

For more on RxJS Transformation Operators, see:

<https://www.learnrxjs.io/operators/combination/>

# RxJS concat Operator

```
import { of } from 'rxjs';
import { concat } from 'rxjs';
// Create observable using of that emits 1, 2
const data1 = of('a','b');
// Create observable using of that emits 3, 4
const data2 = of('x','y');
// Emit values from data1
// when complete subscribe to data2
const example = concat(data1, data2);
// Subscribe to the observable and log the values
const sub = example.subscribe(x => console.log(x));
//Output:
// a
// b
// x
// y
```



Simple example of the **concat** operator

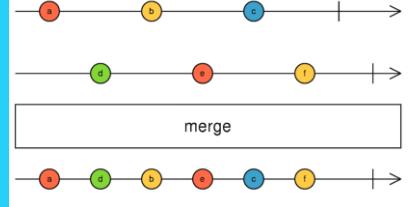
All marble diagrams featured are sourced from ReactiveX© <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

# RxJS merge Operator

```
import { interval, of } from 'rxjs';
import { merge } from 'rxjs';
// Create observable that emits 3 values at 1 per sec
const data1 = interval(1000).pipe(of(a,b,c));
// Create observable that emits 3 values at 1 per 2 sec
const data2 = interval(1250).pipe(of(d,e,f));

// Emit values from data1 or data2 as they arrive
const example = merge(data1, data2);
// Subscribe to the observable and log the values
example.subscribe(x => console.log(x));

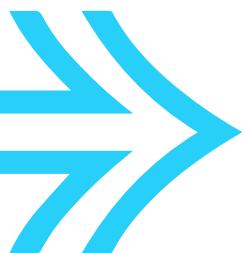
//Output:
// a      - From data1 after 1 second
// d      - From data1 after 1.25 seconds
// b      - From data2 after 2 seconds
// e      - From data2 after 2.5 seconds
// c      - From data1 after 3 seconds (completed)
// f      - From data2 after 3.75 seconds (completed)
```



Simple example of the **merge** operator

All marble diagrams featured are sourced from ReactiveX® <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

## QuickLab 6a – Part 3 - Combination Operators



In this QuickLab you will:

- Use the `interval`, `concat`, `merge` and `take` operators to create multiple **Observables** and concatenate them

QA

# RxJS

## Operators

### Transform Operators



Angular Logo from: <https://angular.io/presskit>

## Observable Transform Operators

RxJS functions that can be used to transform values from Observables

- Transform a value produced by an observable and then emit these through another observable
- They include:

Transform Function	Purpose
<code>map</code>	Take a value and convert it to another, returning the new value
<code>mergeMap/flatMap</code>	Flatten an inner observable but manually control number of inner subscriptions
<code>concatMap</code>	Map values to inner observable, subscribe and emit in order
<code>switchMap</code>	Map to observable, complete previous inner observable, emit values

- For RxJs v6+ these should be imported individually from the rxjs/operators package

For more on RxJS Transformation Operators, see:

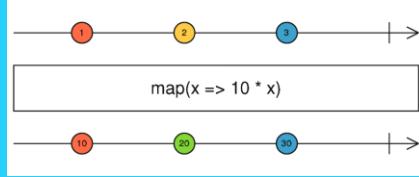
<https://www.learnrxjs.io/operators/transformation/>

# RxJS map Operator

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';
// Create observable using of
const data = of(1, 2, 3);
// Map any incoming values and emit the rhs of arrow fn
// as a new observable
const example = data.pipe(map((x: number) => 10 * x));

// Subscribe to the observable and log the values
example.subscribe(x => console.log(x));

//Output:
// 10
// 20
// 30
```



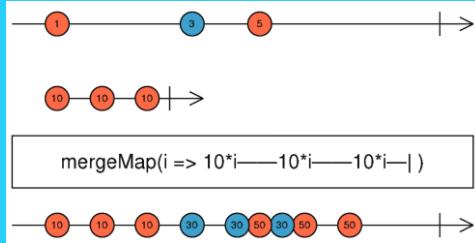
Simple example of the  
**map** operator

All marble diagrams featured are sourced from ReactiveX® <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

# RxJS mergeMap Operator

```
import { of } from 'rxjs';
import { mergeMap } from 'rxjs/operators';
// Create observable using of
const data = of(`Hello`, `Cruel`, `Happy`);
// Map to an inner observable and flatten
const example = data.pipe(mergeMap(val => of(
  `${val} World!`)));
// Subscribe to the observable and log the values
example.subscribe(val => console.log(val));

//Output:
// Hello World!
// Cruel World!
// Happy World!
```



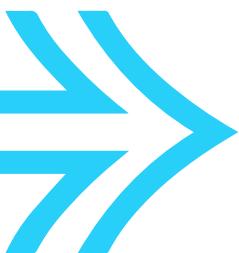
Simple example of the `mergeMap` operator

## NOTE:

Marble diagram does not illustrate this example

All marble diagrams featured are sourced from ReactiveX® <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

## QuickLab 6a – Part 4 - Transformation Operators



In this QuickLab you will:

- Use the `fromEvent` and `map` operators to create an **Observable** and transform the output from it

QA

## RxJS Operators Error Operators



Angular Logo from: <https://angular.io/presskit>

## Observable Error Operators

RxJS functions that can be used to handle error values from Observables

- Error can be handled and, if necessary, data can be retried for
- They are:

Transform Function	Purpose
<code>catch/catchError</code>	Gracefully handle errors in an observable sequence
<code>retry</code>	Retry an observable sequence a specific number of times should an error occur
<code>retryWhen</code>	Retry an observable sequence on error based on custom criteria

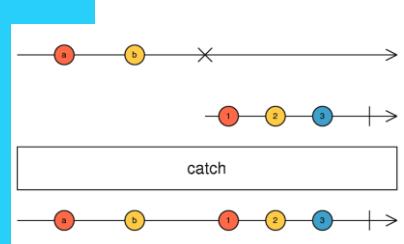
- For RxJs v6+ these should be imported individually from the rxjs/operators package

For more on RxJS Transformation Operators, see:

[https://www.learnrxjs.io/operators/error\\_handling/](https://www.learnrxjs.io/operators/error_handling/)

## RxJS catchError Operator

```
import { of, interval } from 'rxjs';
import { map, catchError } from 'rxjs/operators';
// Create observable using of
const data = of('a', 'b', 'c');
// Gracefully handle error, returning observable
// with error message
const example = data.pipe(
  map(letter => {
    if (letter === 'c') throw `c!`;
    return letter;
}),
  catchError(err => of(1, 2, 3));
// Subscribe to the observable and log the values
example.subscribe(val => console.log(val));
//Output:
// a
// b
// 1
// 2
// 3
```



Simple example of the  
**catchError** operator

All marble diagrams featured are sourced from ReactiveX® <http://reactivex.io/rxjs/manual> licensed under CC 3.0 (<https://creativecommons.org/licenses/by/3.0/>)

```
import { of } from 'rxjs';
import { map, catchError } from 'rxjs/operators';

const data = of('a', 'b', 'c')

const exmaple = data.pipe(
  map(letter => {
    if (letter === 'c') {
      throw `c!`;
    }
    return letter;
  }),
  catchError(err => of(1, 2, 3)));

exmaple.subscribe(x => console.log(x));
```

QA

## Observables in Angular

Where Observables are used in  
Angular Event Emitter



Angular Logo from: <https://angular.io/presskit>

## Observables in Angular

Used as an interface to handle wide range of common asynchronous operations:

- Angular's `EventEmitter` class extends Observable
- AJAX request and responses are handled by Observables in the `HTTP` module
- `Routers` use Observables to respond to user navigation events
- `FormsModule` uses Observables to listen for and respond to user-input events
- `AsyncPipe` use Observables to mark components to be checked for changes

NOTE: `HTTP`, `FormsModule`, `Routers` and `AsyncPipe` use of Observables will be covered in the specific material dedicated to them.

## EventEmitter

A class built-in to Angular

- Used for publishing values from a component through an `@Output`

It extends Observable

- Has an additional `emit()` method so it can send arbitrary values
- Calling `emit()` passes the emitted value to the `next()` method of any subscribed observer

## EventEmitter Example

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

@Component({
  selector: 'emitter-component',
  templateUrl: 'emitter-component.component.html',
})
export class EmitterComponent {
  aBoolean: true;
  @Output() emitSomething = new EventEmitter<any>();
  handle() {
    if (aBoolean) {
      this.emitSomething.emit(true);
    } else {
      this.emitSomething.emit(false);
    }
}
```

## **QuickLab 6b – Observables in Angular**



In this QuickLab you will:

- Examine previously written code to see an example of how Angular extends Observables



## Objectives

- To understand what an Observable is
- To understand what RxJS is
- To be able to use RxJS operators to work with Observables
- To understand how Observables can be used in Angular



Angular Logo from: <https://angular.io/presskit>



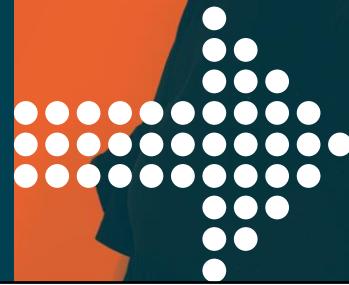
## Template Driven Forms

Building Web Applications using Angular



## Objectives

- To understand the similarities and differences between Template-Driven and Reactive Forms
- To be able to build template-driven forms by:
  - Binding data between the template and component
  - Tracking the changes to and validity of data
  - Using data supplied by forms to provide feedback to users
  - Controlling the submission of data



Angular Logo from: <https://angular.io/presskit>



## TEMPLATE DRIVEN FORMS

**Introduction**

**Comparison to Reactive Forms**

**Model set up**

**Data Flow**



# Introduction

Forms are an integral part of many common applications

- User Log In
- Profile updates/creation
- Other data entry

Angular provides 2 different approaches to handling user input through forms

- Both capture user input events from the template, validate and track data

Reactive Forms	Template Forms
<ul style="list-style-type: none"><li>• More robust</li><li>• Scalable</li><li>• Reusable</li><li>• Testable</li></ul>	<p>Use if forms are key part of application Use if already using reactive patterns for building application</p> <ul style="list-style-type: none"><li>• Easy to add to app</li><li>• Less scalable</li><li>• Useful for adding a simple form to an app</li></ul> <p>Use if form and logic requirements are basic that can be managed on template</p>

## Key Differences Between Reactive and Template Driven Forms

	REACTIVE	TEMPLATE-DRIVEN
Setup (form model)	More explicit, created in component class	Less explicit, created by directives
Data Model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Form Validation	Functions	Directive
Mutability	Immutable	Mutable
Scalability	Low-level API access	Abstraction on top of APIs

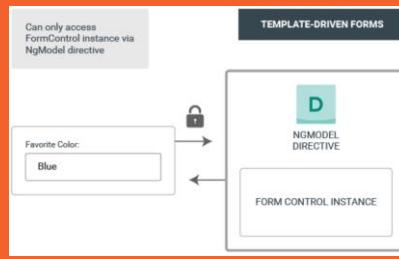
## Commonalities

Both form types share and use some building blocks supplied by Angular

- **FormControl** – tracks the value and validation status of an individual form control
- **FormGroup** – tracks the same values and status for a collection of form controls
- **FormArray** – tracks the same values and status for an array of form controls
- **ControlValueAccessor** – creates a bridge between Angular **FormControl** instances and native DOM elements

# Form Model Setup for Template-Driven Forms

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-template-favourite-colour',
  template: `
    Favourite Colour:
    <input type="text" [(ngModel)]="favouriteColour">
  `
})
export class FavouriteColourComponent {
  favouriteColour: '';
}
```



A Form Model tracks value changes between Angular form elements and input form elements.

Example shows an input field for a single control.

**favouriteColour** from the template is 2-way bound to **favouriteColour** in the component class.

**[(ngModel)]** creates and manages the **FormControl** instance.

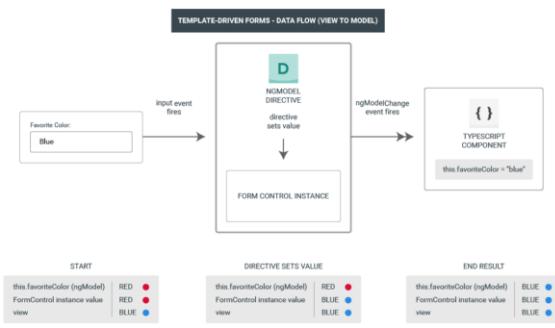
No direct control over form model

# Data Flow – View to Model

Each form element linked to a directive that manages form model directly

Outline data flow from view to model when input value changes:

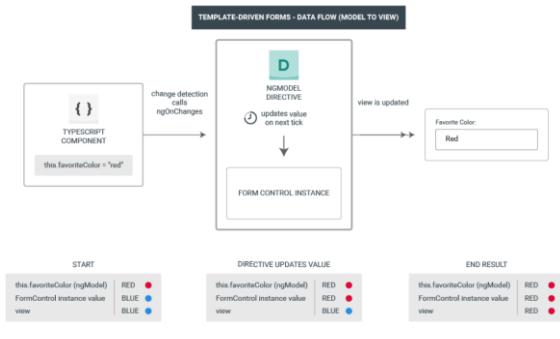
1. User types blue into input element
2. Input element emits an 'input' event
3. Control value accessor attached to input triggers `setValue()` method on `FormControl`
4. `FormControl` emits new value through `valueChanges` observable
5. Subscribers to `valueChanges` receive new value
6. Control value accessor calls `ngModel.viewToModelUpdate` which emits `ngModelChange` event
7. `favoriteColor` property in component is updated to value emitted (due to 2-way data binding)



# Data Flow – Model to View

Each form element linked to a directive that manages form model directly

Outline data flow from model to view when input value changes:



1. **favouriteColor** value updated in component
2. Change detect begins
3. During this, **ngOnChanges** called on **NgModel** because value of one of its inputs has changed
4. **ngOnChanges()** queues an async task to set value for internal **FormControl** instance
5. Change detection completes
6. On next tick, task to set **FormControl** instance value executed
7. **FormControl** emits latest value through **valueChanges** observable
8. Subscribers to **valueChanges** receive new value
9. Control value accessor updates form input element with latest **favouriteColor** value



# DEVELOPING TEMPLATE DRIVEN FORMS

**Skills**

**Steps for development**

**Module Setup**

**Creating the Form Template**

**Data binding with the form**



# Developing Template-Driven Forms

The following steps are usually taken to create a template-driven form:

1. Build an Angular form with a component and template
2. Use `ngModel` to create 2-way data bindings for read/write of input control values
3. Track state changes and validity of form controls
4. Provide visual feedback using special CSS classes that track state of controls
5. Display validation errors and enable/disable form controls
6. Submit the data

Requires design skills and Angular support for:

- 2-way data binding
- Change tracking
- Validation
- Error Handling

Can build almost any form with this method

Additionally possible to:

- Lay out controls creatively;
- Bind them to data;
- Specify validation rules and display errors;
- Conditionally enable/disable specific controls
- And much more...

## Module Setup

Creating a form requires the import of the Angular **FormsModule** into the Module the form is to belong to

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

@NgModule({
  imports: [FormsModule]
}) export class SomeModule
```

This gives us access to the directives we need to build an interactive form

# 1. Create the Form Template

A form template is little more than HTML combined with Angular directives we are already partly familiar with

```
<form>
  <label for="firstname">First name</label>
  <input type="text" id="firstname" name="firstname" required>

  <label for="surname">Surname</label>
  <input type="text" id="surname" name="surname" required>

  <label for="age">Age</label>
  <input type="number" id="age" name="age">

  <button type="submit">Submit</button>
</form>
```

There's nothing particularly "Angular" about this form. Until we furnish it with some Angular directives

## 2. Data Bind with ngModel

**ngModel** allows us to create two way data binding between our form control components and the model

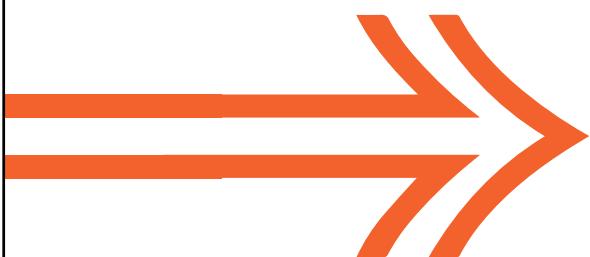
```
<input type="number" name="age" [(ngModel)]="user.age">
```

Which we can use in the model or in the view (usually for diagnostic purposes)

```
{{user.age}}
```

**ngModel** will implicitly attach an **NgForm** directive to the **<form>** tag if the input is used as part of a form

## **QuickLab 7 – Template- Driven Forms – Parts 1-3**



In these parts of the QuickLab, you will:

- Import the Modules needed to work with template driven forms
- Add a form to a template and bind the data using the ngModel directive

(Up to the end of Part 3)



## DEVELOPING TEMPLATE DRIVEN FORMS

**Validity, State and ngModel**  
**Visual feedback using CSS classes**



## 3. Tracking control state and validity with ngModel

Leverages **HTML5** validation

**ngModel** allows access to information about the status of each input

- Has the control been touched?
- Has the control's value changed?
- Is the entered value valid?

**ngModel** updates the control with special Angular CSS classes that reflect the state:

- Adding a template reference to return the element gives access to the **className(s)** that are applied to it – done through the use of **#templatereferencevariablename**
- **className(s)** can be leveraged to change the appearance of the control

State	Class if true	Class if false
Control has been visited	<b>ng-touched</b>	<b>ng-untouched</b>
Control's value has changed	<b>ng-dirty</b>	<b>ng-pristine</b>
Control's value is valid	<b>ng-valid</b>	<b>ng-invalid</b>

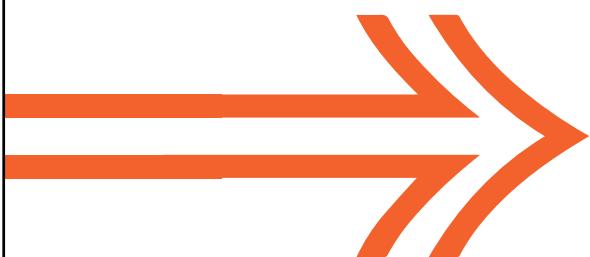
## 4. Provide visual feedback using special CSS classes

```
input.ng-dirty.ng-invalid {  
    border: 2px solid red  
}  
  
input.ng-dirty.ng-valid {  
    border: 2px solid green  
}
```

Visual feedback can be easily provided

- Defining some CSS classes that correspond to the input value and its validity
- Eg. If the user has visited the field and the value is invalid

## **QuickLab 7 – Template- Driven Forms – Parts 4-5**



In these parts of the QuickLab, you will:

- Examine how Angular tracks changes to fields and their assesses their validity
- How validity can be used to provide visual feedback via CSS  
(Up to the end of Part 5)



## DEVELOPING TEMPLATE DRIVEN FORMS

Displaying validation errors  
Submitting forms



## 5. Display validation errors

```
<input  
  type="text"  
  name="name"  
  [(ngModel)]="user.name"  
  #tempRef="ngModel"  
>  
  
<div [hidden]="tempRef.valid">  
  Name is invalid  
</div>
```

The values available because of the template reference variable being set to `ngModel` can be used to conditionally display markup

- Eg. a div explaining that the name is invalid

We can read values of the template by setting a **template reference** variable to `ngModel`

The values can be accessed to decide the `hidden` attribute or in an `*ngIf`

The `NgModel` directive has an `exportAs` property that is equal to `ngModel` and hence why we use that in the template reference variable

## 6. Submitting the form

```
<form  
  #myForm="ngForm"  
  (ngSubmit)="onSubmit()">  
  ...  
  <button  
    type="submit"  
    [disabled]="!myForm.form.valid"  
  >  
    Submit  
  </button>  
</form>
```

Handled through the `ngSubmit` event emitter provided by the `NgForm` directive

- This can then be linked to a method in the Component class

The `NgForm` directive creates a top-level `FormGroup` instance binding it to track the aggregated form value and validation status

- This can then be used to do things like enabling/disabling the submit button if the form is invalid

## **QuickLab 7 – Template- Driven Forms – Parts 6-8**



In these parts of the QuickLab, you will:

- Use template reference variables, ngclasses and the hidden attribute to display validation messages
- Examine form submission
- Use data-binding to set data in the component class

QA

## TESTING TEMPLATE- DRIVEN FORMS



Due to their nature, it is not possible to test template-driven forms using the testing tools discussed so far

End-to-End (e2e) testing has to be employed

- Uses Protractor if using the Angular CLI embedded testing tools





## Objectives

- To understand the similarities and differences between Template-Driven and Reactive Forms
- To be able to build template-driven forms by:
  - Binding data to the template and component
  - Tracking the changes to and validity of data
  - Using data supplied by forms to provide feedback to users
  - Controlling the submission of data



Angular Logo from: <https://angular.io/presskit>



# Reactive Forms

Building Web Applications using Angular





## Objectives

- To be able to build and use reactive forms by being able to:
  - Generate and import a new form controls
  - Register controls in a template
  - Display form control values
  - Group related form controls
  - Associate the FormGroup model with a view
  - Nest a form group
  - Group nested forms in a template
  - Use FormBuilder to create forms
  - Controlling form submission



Angular Logo from: <https://angular.io/presskit>



## Reactive Forms

**Introduction to Reactive Forms  
Comparison to Template-Driven  
forms  
Data Flow explanation**



# Introduction

Forms are an integral part of many common applications

- User Log In
- Profile updates/creation
- Other data entry

Angular provides 2 different approaches to handling user input through forms

- Both capture user input events from the template, validate and track data

Reactive Forms	Template Forms
<ul style="list-style-type: none"><li>• More robust</li><li>• Scalable</li><li>• Reusable</li><li>• Testable</li></ul>	<p>Use if forms are key part of application Use if already using reactive patterns for building application</p> <ul style="list-style-type: none"><li>• Easy to add to app</li><li>• Less scalable</li><li>• Useful for adding a simple form to an app</li></ul> <p>Use if form and logic requirements are basic that can be managed on template</p>

## **Key Differences Between Reactive and Template Driven Forms**

	<b>REACTIVE</b>	<b>TEMPLATE-DRIVEN</b>
Setup (form model)	More explicit, created in component class	Less explicit, created by directives
Data Model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Form Validation	Functions	Directive
Mutability	Immutable	Mutable
Scalability	Low-level API access	Abstraction on top of APIs

## Commonalities

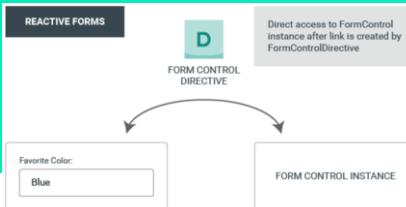
Both form types share and use some building blocks supplied by Angular

- **FormControl** – tracks the value and validation status of an individual form control
- **FormGroup** – tracks the same values and status for a collection of form controls
- **FormArray** – tracks the same values and status for an array of form controls
- **ControlValueAccessor** – creates a bridge between Angular **FormControl** instances and native DOM elements

# Form Model Setup for Reactive Forms

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

@Component({
  selector: 'app-template-favourite-colour',
  template: `
    Favourite Colour:
    <input type="text" [formControl]="favColCtrl">
  `
})
export class FavouriteColourComponent {
  favColCtrl = new FormControl('');
}
```



A Form Model tracks value changes between Angular form elements and input form elements.

Example shows an input field for a single control.

Form model is explicitly defined in component class.

Reactive form directive (**FormControl**) links existing **FormControl** to specific form element in view using **ControlValueAccessor** instance.

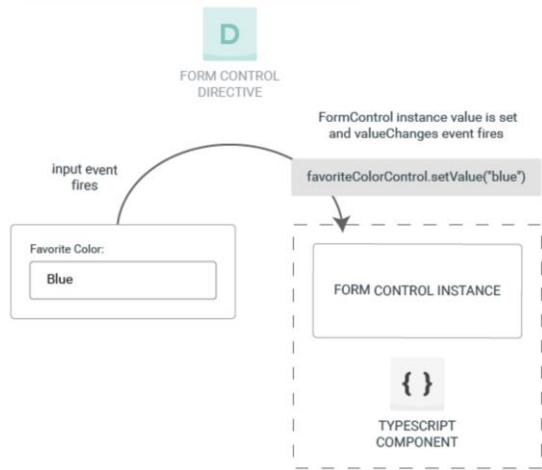
# Data Flow – View to Model

Each form element linked to a directive that manages form model directly

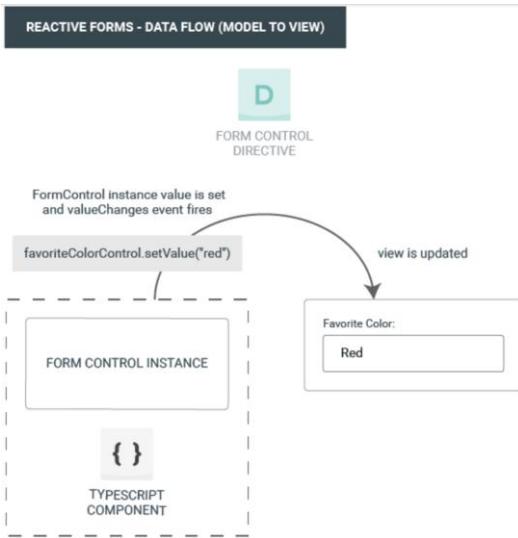
Outline data flow from view to model when input value changes:

1. User types blue into input element
2. Input element emits an 'input' event
3. Control value accessor listening for events on form input element immediately relays new value to `FormControl`
4. `FormControl` emits new value through `valueChanges` observable
5. Subscribers to `valueChanges` receive new value

REACTIVE FORMS - DATA FLOW (VIEW TO MODEL)



## Data Flow – Model to View



Each form element linked to a directive that manages form model directly

Outline data flow from model to view:

1. User calls `favColCtrl.setValue()` which updates `FormControl` value
2. `FormControl` emits new value through `valueChanges` observable
3. Subscribers to `valueChanges` receive new value
4. Control value accessor on form input element updates element with new value



## Developing Reactive Forms - Form Controls

**Skills needed**

**Module Setup**

**Form Controls**

**Creating a Form Control**

**Using a Form Control in a template**



## Developing Reactive Forms

Reactive forms manage state of a form at any given time explicitly and immutably

- Each change to form state returns a new state
- Maintains integrity of the model between changes

Built around observable streams

- Form inputs and values provided as streams of input values, accessed synchronously

Data is consistent and predictable when requested (good for testing)

## Developing Reactive Forms

The following skills are required to create a reactive form:

- Register the reactive forms module
- Generate and import new form controls
- Register control in the template
- Managing the control values
- Grouping the form controls
- Creating nested form groups
- Updating the model
- Using Angular's FormBuilder
- Validating the form
- Dynamic controls using Form Array

## Module Setup

Creating a form requires the import of the Angular ReactiveFormsModule into the Module the form is to belong to

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

@NgModule({
  imports: [ReactiveFormsModule]
}) export class SomeModule
```

This gives us access to the directives we need to build an interactive form

## Generating and Importing a new form control

`FormControl` class instances need to be included in a Component

`FormControl` class is the basic building block for Reactive Forms

Registering a single form control needs the `FormControl` class importing into the Component

- Use the class to create a new instance of `FormControl`

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

@Component({
  selector: 'app-myreactiveform',
  templateUrl: './myreactiveform.component.html'
})
export class MyReactiveForm {
  myFormControl = new FormControl('');
}
```

## Registering the control in the template

```
<fieldset>
  <label for="name">Name: </label>
  <input
    type="text"
    [FormControl]="myFormControl"
  >
</fieldset>
```

A `FormControl` in the component needs to be associated with a form control element in the template

`FormControlDirective` in the `ReactiveFormsModule` supplies `FormControl` binding

Template binding syntax means form control is now registered to the input element

Form control and the element communicate with each other

- View reflects changes in the model
- Model reflects changes on the form

## Managing Control Values - Displaying a value

Values can be displayed by:

- Through `valueChanges` observable

Listen for changes in the `template` using `AsyncPipe` or in `component` using `subscribe()` with the `value` property

```
<p>Value: {{myFormControl.value}}</p>
```

## Managing Control Values - Replacing a value

Can be done programmatically using `setValue()`

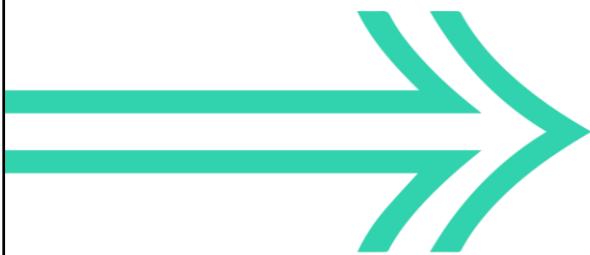
- Method that updates the value of a form control and validates it

```
//in component
updateVal() {
  this.myFormControl.setValue('newVal');
}
```

- Need something to call the method in the template

```
//in template
<button (click)="updateVal()">Update</button>
```

## **QuickLab 8 – Reactive Forms – Parts 1-4**



In these parts of the QuickLab, you will:

- Add the required Modules to work with Reactive forms
- Create FormControl for each field in the component class
- Bind the FormControl to the template and display the value

(Up to the end of Part 4)



## **Developing Reactive Forms - Form Groups**

### **Adding Form Controls to Form Groups**

### **Nesting Form Groups**



# Grouping Form Controls

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
@Component({
  selector: 'app-myreactiveform',
  templateUrl: './myreactiveform.component.html'
})
export class MyReactiveFormComponent {
  myReactiveForm = new FormGroup({
    myFormCtrl1: new FormControl(''),
    myFormCtrl2: new FormControl(''),
  });
}
/* This FormGroup instance provides its model as an object
reduced from the values of each control in the group. */
/* It has the same properties (value, untouched, etc) and
methods (setValue()) as a form control instance */
```

**FormControl** instances can not be used on a form without grouping them.

A **FormGroup** allows this to happen by taking an object of **FormControls**

- Each control included as part of a form group is tracked by name when creating the form group

The **FormGroup** class has to be imported

A name must be given for the form group

## Associating the FormGroup model and view

```
<form [formGroup]="myReactiveForm">
  <label for="myFormCtrl1">FormControl1: </label>
  <input type="text"
    formControlName="myFormCtrl1"
    required
  >
  <label for="myFormCtrl2">FormControl2: </label>
  <input type="text"
    formControlName="myFormCtrl2"
    required
  >
</form>

<p>
  myFormCtrl1 values is:
  {myReactiveForm.value.myFormCtrl1}
</p>
```

**FormGroup** tracks status and changes for each control named in it

- If one of control changes, parent control also emits a new status or value change

Model for group maintained from its members

- After defining the model, update the template to reflect the model value

## Creating Nested form groups

```
...
myForm = new FormGroup({
  myFormCtrl1: new FormControl(''),
  myFormCtrl2: new FormControl(''),
  mySubFormGp: new FormGroup({
    subFmCtrl1: new FormControl(''),
    subFmCtrl2: new FormControl(''),
  })
}) ;
...
...
```

As forms become more complex it is often desirable to split the form into smaller chunks

- Some groups of information naturally associate with each other

Nested form groups are simply a **FormGroup** contained within another **FormGroup**

- This allows large form groups can be broken down into a set of smaller ones

## Grouping the nested form in the template

```
<fieldset formGroupName="mySubFormGp">
  <label
    for="subFmCtrl11">SubFmCtrl1:
  </label>
  <input
    type="number"
    formControlName="subFmCtrl11">
  <label for="subFmCtrl12">SubFmCtrl2:
  </label>
  <input
    type="text"
    formControlName="subFmCtrl12">
</fieldset>

<p>
  subFmCtrl11: {{parentGroup.value.mySubFormGp.subFmCtrl11}}
</p>
```

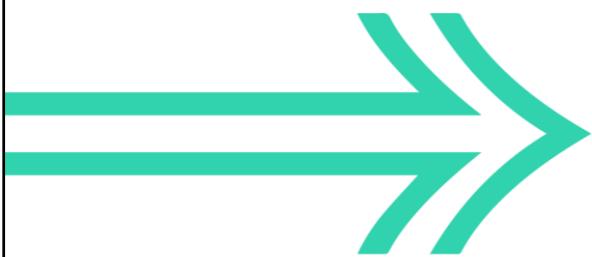
The enclosing element should possess the form group's name

All form controls can then be named as usual

- Angular will deal with the associations between the template and the component!

Accessing the sub-group values on the form in the template can be done by using the pattern shown

## **QuickLab 8 – Reactive Forms – Parts 5-8**



In these parts of the QuickLab, you will:

- Add a FormGroup to the class
- Associate this group with the template using the `formGroup` directive
- Nest a group inside a group
- Add the nested group to the template

(Up to the end of Part 8)



## Developing Reactive Forms - Form Groups

### Updating the model



# Partial Model Updates

Sometimes desirable to only update parts of a form group that contains multiple controls

Angular provides the methods `setValue()` and `patchValue()` to update the model values

## `setValue()`

Sets a new value for an individual control  
Strictly adheres to structure of form group  
Replaces entire value for the control

## `patchValue()`

Replaces any properties defined in object that have changed in the form model

`patchValue()` can be used to update only parts of the model, perhaps using a button

```
// In Component
...
updateValues() {
  this.myReactiveForm.patchValue({
    myFormCtrl1: 'new value',
    mySubFormGp: {
      subFmCtr2: 'new value'
    }
  });
}
...
<button(click)="updateValues()">
  Update
</button>
```

QA



## **Developing Reactive Forms - Form Builder**

**Using Form Builder**

**Creating Groups**

**Adding Form Controls**



## Using FormBuilder

The process described so far can be laborious when dealing with multiple forms  
Angular provides a **FormBuilder** service that has useful methods for generating controls  
It requires the import of the **FormBuilder** class from `@angular/forms`

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

This service is then injected into the constructor of the form's component class

```
...
constructor(private fb: FormBuilder) {}
```

# Generating Form Controls with Form Builder

**FormBuilder** service provides 3 methods:

Method	Description
<b>group()</b>	Constructs a new <b>FormGroup</b> instance
<b>control()</b>	Constructs a new <b>FormControl</b> with the given state, validators and options
<b>array()</b>	Constructs a new <b>FormArray</b> from the given array of configurations, validators and options

The form could be re-written as shown, using the **FormBuilder** injected into the component

```
import { Component } from
'@angular/core';
import { FormBuilder } from
'@angular/forms';
@Component({ ... })

export class MyFormComponent {
  constructor(private fb: FormBuilder) {}
  myForm = this.fb.group({
    myFormCtrl1: [''],
    myFormCtrl2: [''],
    mySubFormGp: this.fb.group({
      subFmCtrl1: [''],
      subFmCtrl2: ['']
    })
  });
}
```

228

## Submitting the form (same as Template Driven forms)

Handled through the `ngSubmit` event emitter provided by the `NgForm` directive

- This can then be linked to a method in the Component class

The `NgForm` directive creates a top-level `FormGroup` instance binding it to track the aggregated form value and validation status

- This can then be used to do things like enabling/disabling the submit button if the form is invalid

```
<form
  #myForm="ngForm"
  (ngSubmit)="onSubmit()" >
...
<button
  type="submit"
  [disabled]="!myForm.form.valid"
>
  Submit
</button>
</form>
```

## **QuickLab 8 – Reactive Forms – Part 9**



In this part of the QuickLab, you will:

- Use FormBuilder as an alternative to FormControls and FormGroups



## Objectives

- To be able to build and use reactive forms by being able to:
  - Generate and import a new form controls
  - Register controls in a template
  - Display form control values
  - Group related form controls
  - Associate the FormGroup model with a view
  - Nest a form group
  - Group nested forms in a template
  - Use FormBuilder to create forms
  - Controlling form submission



Angular Logo from: <https://angular.io/presskit>

## Appendix – Observe Control Changes

Angular does not call **ngOnChanges** when the user modifies a **FormControl**

- Changes can be monitored by subscribing to one of the form control properties that raise an event

Such as **valueChanges** – returns an RxJS Observable

Add methods to log the changes:

```
// Component excerpt
prop1ChangeLog: string[] = [];
logProp1Change() {
  const prop1Control = this.myForm.get('prop1');
  prop1Control.valueChanges.forEach(
    (value: string) => this.prop1ChangeLog.push(value)
  )
}
```

Call method in the constructor – outputting the **prop1ChangeLog** in the template will show all keystrokes

Interpolation binding is the easier way to display name changes in the UI.

Subscribing to an observable form control property is useful as a trigger for application logic within the component class.

## Appendix – Dynamic Forms (1)

Reactive Forms created based on metadata that describes the business object model  
Can use **FormGroup** to dynamically render forms with different control types and validation  
Allows creation of forms on-the-fly without changing the application code

RECIPE...

1. Define an object model that can describe all scenarios needed by the form functionality
2. Derive classes based on the types of inputs required on the form
  - Idea is that form will be bound to specific input types and render the appropriate controls dynamically
3. Create a service to convert the input types into a **FormGroup**
  - Form group consumes the meta data from the object model and allow specification of default values/validation
4. Create components that represent the dynamic form and a parent component for the form
  - Using **\*ngSwitch** directive can determine which of the components to display

## Appendix – Dynamic Forms (2)

RECIPE (ctd.)...

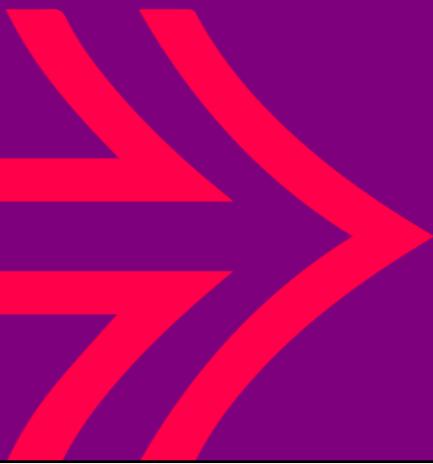
5. The wrapping component expects the input types in the form of an array bound to an `@Input`
  - These can be retrieved by the service defined
  - Key point is that inputs are controlled entirely through objects returned from the service  
Input maintenance is simply a matter of adding, updating and removing objects from the inputs array
6. Use the service in the component that will contain the dynamic form, using it to set the array of inputs to be used on the form

A worked example of this can be found at: <https://angular.io/guide/dynamic-form>



# Form Validation

Building Web Applications using Angular





## Objectives

- To understand how to validate template-driven forms
- To understand how to validate reactive forms
- How to use Validator functions
- How to use built-in Validators
- To be able to create Custom Validators
- To understand how cross-field validation works
- To understand how Async Validation works



Angular Logo from: <https://angular.io/presskit>



## **Template-Driven Form Validation**

**Validating a template-driven form  
Using HTML5 validation**



Angular Logo from: <https://angular.io/presskit>

## Template-Driven Validation

Add HTML5 form validation attributes to the template

- Angular directives match the attributes with provided validator functions

Angular runs validation every time form control value changes

- Generates:

- List of validation errors resulting in INVALID status
- Null resulting in VALID status

Can export `ngModel` to a local template variable to allow inspection

## Template-Driven Validation

```
<input  
  name="myInput"  
  required  
  minlength="4"  
  [(ngModel)]="myModel.myInput"  
  appForbiddenValue="banned"  
  #myInput="ngModel"  
>  
  
<div *ngIf="myInput.invalid && (myInput.dirty ||  
myInput.touched)">  
  <p *ngIf="myInput.errors.required">Field required</p>  
  <p *ngIf="myInput.errors.minLength">  
    Must be 4 characters long  
  </p>  
  <p *ngIf="myInput.errors.forbiddenValue">  
    Value cannot be banned  
  </p>  
</div>
```

`<input>` has HTML attributes `required` and `minlength` and custom validator `forbiddenValue`  
`#ngModel` exports `NgModel` to local variable allowing checking of control states (eg. `dirty`)  
`*ngIf` on `<div>` uses control states to decide if to show at all  
Each `<p>` can present custom messages for validation error present



## **Reactive Form Validation Validator Functions Built-In Validators**



Angular Logo from: <https://angular.io/presskit>

## **Reactive Form Validation**

Source of truth for form is component class

- Validator functions applied directly to form controls in component class
- Called when value of control changes

# Validator Functions

Two types of validator functions: sync and async validators:

Sync Validators	Async Validators
<p>Take control instance and immediately return either:</p> <ul style="list-style-type: none"><li>• Set of validation errors</li><li>• null</li></ul> <p>Can pass these in as second argument when instantiating a <code>FormControl</code></p>	<p>Take control instance and returns a Promise or Observable that either:</p> <ul style="list-style-type: none"><li>• Emits set of validation errors</li><li>• null</li></ul> <p>Can pass these in as third argument when instantiating a <code>FormControl</code></p>

## Built-in Validators

Angular provides a set of built-in validators, some mirroring HTML5 validation attributes:

Validator Name	Validating Action
<code>min</code>	Control's value must be greater than or equal to the number specified
<code>max</code>	Control's value must be less than or equal to the number specified
<code>required</code>	Control must have a value (ie. Not empty)
<code>requiredTrue</code>	Control's value must be <code>true</code> – useful for checkboxes
<code>email</code>	Control's value must pass an email validation test
<code>minLength</code>	Control's value must be greater than or equal to the specified length – provided by default if HTML5's <code>minlength</code> attribute is used
<code>maxLength</code>	Control's value must be less than or equal to the specified length – provided by default if HTML5's <code>maxlength</code> attribute is used
<code>pattern</code>	Control's value must match a regex pattern - provided by default if HTML5's <code>pattern</code> attribute is used

## Built-in Validators

Other built-in validators are:

Validator Name	Validating Action
<code>nullValidator</code>	Performs no operation
<code>compose</code>	Makes multiple validators into a single function – returns the union of the individual error maps for the provided control
<code>composeAsync</code>	Makes multiple async validators into a single function – returns the union of the individual error maps for the provided control

## Adding Validators to Controls

```
import { FormControl, FormGroup, Validators } from  
'@angular/forms';  
  
...  
  
this.myForm = new FormGroup({  
  prop1: new FormControl(  
    this.myDataModel.property1,  
    Validators.requiredTrue  
  ),  
  prop2: new FormControl(  
    this.myDataModel.property2,  
    [Validators.required, Validators.minLength(10)]  
  )  
});  
...
```

When using a **FormGroup** and explicit **FormControl**s, add validators to controls by:

1. Supplying a single Validator
2. Supplying an array of Validators

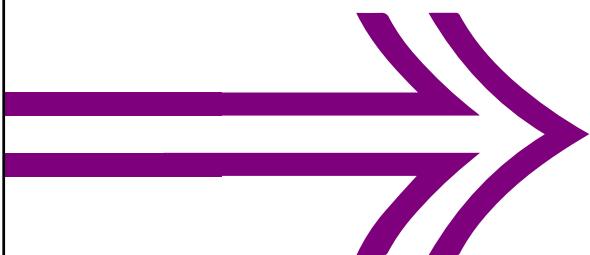
## Adding Validators to Controls in FormBuilder

```
import { FormBuilder, Validators } from  
'@angular/forms';  
...  
constructor(private fb: FormBuilder) {}  
...  
this.myForm = this.fb.group({  
  prop1: [  
    this.myDataModel.property1,  
    Validators.requiredTrue  
  ],  
  prop2: [  
    this.myDataModel.property2,  
    [Validators.required, Validators.minLength(10)]  
  ]  
});  
...
```

When using **FormBuilder**, Add validators to controls by:

1. Supplying a single Validator
2. Supplying an array of Validators

## **QuickLab 9a – Reactive Form Validation – Part 1**



In this part of the QuickLab, you will:

- Examine the built-in validators provided by Angular for Reactive Forms



# **Reactive Form Validation**

## **Custom Validator Functions**

### **Writing a Custom Validator Using the Custom Validator**



Angular Logo from: <https://angular.io/presskit>

## Custom Validators – Reactive Forms

- Built-in validators don't always do what you need them to
- Custom validators can be created – usually in their own **.directive.ts** file and imported for use

The following code would create a custom validator to restrict strings to a given pattern

```
// forbidden-string.directive.ts

export function forbiddenStringValidator(stringRe: RegExp): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} => {
    const forbidden = stringRe.test(control.value);
    return forbidden ? {'forbiddenString': {value: control.value}} : null;
  });
}
```

## Custom Validator - Explanation

```
export function forbiddenStringValidator(stringRe: RegExp): ValidatorFn {...}
```

Function actually a factory that takes a regex to detect a given forbidden string (or pattern)

- Returns a validator function

```
export function forbiddenStringValidator(stringRe: RegExp): ValidatorFn {  
  return (control: AbstractControl): ValidationErrors => {...}  
}
```

- Function returned takes form control of type **AbstractControl** as argument  
**AbstractControl** is base class of **FormControl**, **FormArray** and **FormGroup**
- Function returned itself returns an object (with a key value pair)  
Key contains name of the error, value is always **true**  
If validation passes, **null** is returned

## Custom Validator - Explanation

Logic inside returned function:

- Creates a **boolean** as the result of a test of the **regex supplied** against the **control value**
- It returns either the required **error object** or **null**

```
export function forbiddenStringValidator(stringRe: RegExp): ValidatorFn {
  return (control: AbstractControl): ValidationErrors => {
    const forbidden = stringRe.test(control.value);
    return forbidden ? {'forbiddenString': {value: control.value}} : null;
  };
}
```

Note that anything can be validated – not just strings against regular expressions

- Eg. 2 numbers could be passed into the Validator function to create range validation

## Using a Custom Validator in Reactive Forms

The Custom Validator directive need not be **declared** in the Module or be a class

- Can simply be an **exported** function from a file

The Validator function needs to be **imported** into the Component that the form lives in

```
import { validatorFunctionName } from '/path/to/directive/directive-name';
```

The Validator function can then be added to the Control either as an individual or as part of the array of validators

```
this.myForm = this.fb.group({
  prop1: [this.myDataModel.property1, validatorFunctionName(args)],
  prop2: [
    this.myDataModel.property2,
    [Validators.required, validatorFunctionName(args)]
  ]
});
```

## Accessing Values in the template

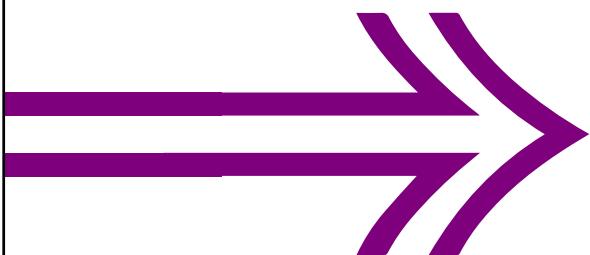
When using a **FormBuilder**, values for individual controls can be accessed using the **get()** function.

It called on the **FormGroup**'s name and the control to access is passed as a string.

Any further properties can then be chained such as invalid, dirty, errors:

```
<form [formGroup]="myForm">
  <input type="text" name="prop2" formControlName="prop2">
  <div *ngIf="myForm.get('prop2').invalid">
    <p *ngIf="myForm.get('prop2').errors.required">A value is required</p>
    <p *ngIf="myForm.get('prop2').errors.validatorFunctionName">Not allowed</p>
  </div>
</form>
```

## **QuickLab 9a – Reactive Form Validation – Part 2**



In this part of the QuickLab, you will:

- Create a custom validator to check that a number is within a specified range



## Template-Driven Form Validation

### Custom Validator Directives

**Creating a Validator Directive**

**Applying the Custom Validation to the template**



Angular Logo from: <https://angular.io/presskit>

## Custom Validators – Template-Driven Forms

```
...
@Directive({
  selector: '[validatorFnName]',
  providers: [{provide: NG_VALIDATORS,
    useExisting: ValidatorDirective,
    multi: true}]
})
export class ValidatorDirective implements
Validator {
  @Input(validatorFnName) forbidden: any;
  validatorFnName(control: AbstractControl):
  {[key: string]: any} | null {
    // Validation logic to return object
    return null;
  }
}
```

Template-Driven forms can use Custom Validators too

No direct access to `FormControl` so can't pass validator in

- Have to add directive to template instead

Directive class has to wrap validator function

- Angular recognizes directive's role in validation because directive needs to register itself with the `NG_VALIDATORS` provider

→ `NG_VALIDATORS` is a provider with extensible collection of validators

256

## Custom Validators – Template-Driven Forms

- Can then add selector as attribute to field to validate, supplying any required data

```
...<input id="inputId" name="inputName" validatorFnName="args">
...
```



## **Other Validation Techniques**

**Cross-field Validation**

**Async Validation**



Angular Logo from: <https://angular.io/presskit>

## Cross-Field Validation

This can be added to evaluate a number of controls in a single custom validator

Validation performed in common ancestor control

- Could be **FormGroup** or **FormBuilder**
- Validation then can be measured on form (for templates) or on **FormGroup** (for reactive)

Templates-Driven forms use Directives as shown previously

Reactive Forms use functions as shown previously

## Async Validation

Have counterparts to synchronous validator interfaces `ValidatorFn` and `Validator`

- `AsyncValidatorFn`
- `AsyncValidator`

Both must return a `Promise` or an `Observable`

Observable returned must be finite – must complete at some point

Pipe through a filtering operator such as `first`, `last`, `take` or `takeUntil`

Must happen after synchronous validation has completed and has been successful

- Avoids potentially expensive async validation processes if a more basic validation fails

## Async Validation

Once begun, form control enters pending state

- Can inspect control's pending property and use it to give visual feedback about ongoing validation

Common pattern is to show a spinner whilst async validation is being performed

Custom async validators can be written as required



## Objectives

- To understand how to validate template-driven forms
- To understand how to validate reactive forms
- How to use Validator functions
- How to use built-in Validators
- To be able to create Custom Validators
- To understand how cross-field validation works
- To understand how Async Validation works



Angular Logo from: <https://angular.io/presskit>

QA

## Testing Reactive Forms

**Testing Field Validity**

**Testing Field Errors**

**Testing Form Validity and  
readiness for submission**

**Testing Custom Validators**



Jasmine

Angular Logo from: <https://angular.io/presskit>

# Testing Field Validity

```
...
beforeEach(() => {
  fixture =
 TestBed.createComponent(AppComponent);
  app = fixture.debugElement.componentInstance;
  fixture.detectChanges();
});

it(`should test validity of field`, () => {
  const field = app.form.get('field');
  field.setValue(`Some value`);
  expect(field.value).toBe(`Some value`);
  expect(field.valid).toBe(true);
});
...
```

Field validity can be tested as a class-style test

Need to get the control from the form:

- Can inspect its value
- Can change its value
- Can inspect its error state

Component set up using `fixture` and `componentInstance` of `debugElement` in normal way

Field to be inspected lifted from form group using `component.form.get('fieldName')`  
`setValue()` can be used to change the value

`valid` can be used to inspect validity

264

## Testing Field Errors

```
...
it(`should test validity of field`, () => {
  const field = app.form.get('field');
  field.setValue(``);
  expect(field.errors[`required`]).toBe(true);
  expect(field.hasError(`required`)).toBe(true);
});
...

```

Errors are generated by Validators applied to field

Can be accessed through:

- Control's `error` object
- Use of `hasError()`

# Testing Form Validity

Testing the validity property of a form is fairly simple:

- Make every control on the form valid
- Assert that the form's validity status is true

```
...
  beforeEach(() => {
    fixture =
    TestBed.createComponent(AppComponent);
    app = fixture.debugElement.componentInstance;
    fixture.detectChanges();
  });

it(`should test validity of field`, () => {
  const field1 = app.form.get('field1');
  const field2 = app.form.get('field2');
  field1.setValue(`Some valid value`);
  field2.setValue(`Some valid value`);

  expect(app.form.valid).toBe(true);
});
...
```

# Testing Custom Validators

```
...
describe(`test validator`, () => {
  const validatorFunc =
actualValidator([..args]);
  const control: { value: any } = {value: ``};
  let result: ValidationErrors;
  it(`should return correctly`, () => {
    control.value = `Good value`;
    result = validatorFunc(control as
AbstractControl);
    expect(result).toBeNull();
    control.value = `Bad value`;
    result = validatorFunc(control as
AbstractControl);
    expect(result).toBe(true);
  });
});
...

```

Can be written in pure Jasmine Tests

- No Angular testing utilities needed
- Need to obtain the `ValidatorFn` from the file by setting a constant to a function call with the appropriate arguments

Need a `control` value (an object with a `value` key of `any`) and a `result` (of type `ValidationErrors`)

- Passed to validation function as `AbstractControl`
- Can set value to test

QA

## Testing Reactive Forms

**Testing Field Validity**

**Testing Field Errors**

**Testing Form Validity and  
readiness for submission**

**Testing Custom Validators**

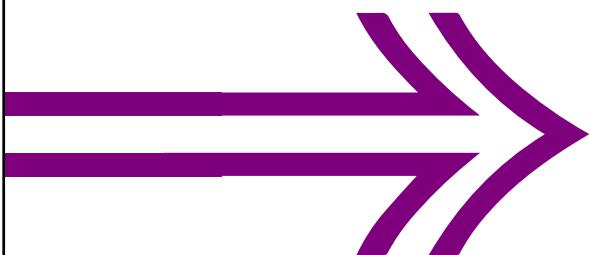


Jasmine

Angular Logo from: <https://angular.io/presskit>

## **QuickLab 9b –**

### **Testing Reactive Form Validation**



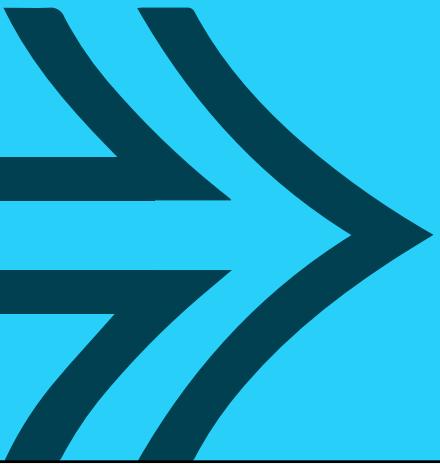
In this QuickLab, you will:

- Set up a test environment for Reactive forms
- Test individual field validity
- Test form validity conditions
- Write tests to ensure a custom validator works as expected



## Services

Building Web Applications using Angular





## Objectives

- To understand when and how to create a Service and provide them in the appropriate place
- To be able to create tests for Services
- To be able to use Services through dependency injection
- To be able to make Services interact with REST APIs using Observables



Angular Logo from: <https://angular.io/presskit>



## Services

### Simple example



Angular Logo from: <https://angular.io/presskit>

# Introduction to Services

Building blocks to provide any value, function or feature that an app may need

- Usually a JavaScript class with a narrow responsibilities – does something specific and does it well

Distinguished from Components

- Increases modularity and reusability
- Helps keeps Components view-focused making them lean and efficient

Component's only job is to provide a UI and nothing else using methods and properties for data-binding

Services are connected to Components who delegate tasks such as:

- Retrieving data from the server;
- Validating user input
- Logging directly to the console

# Introduction to Services

Services are injectable into Components

- Tasks can be accessed by any Component who have the Service injected into them
- Different providers of the same type of service can be injected as appropriate

Principles not enforced by Angular

- However, easy to follow them

Application logic easy to factor into Services

Services made available to Components through Dependency Injection

Services allow easier unit-testing by allowing mock-data to be provided to Components

- Abstracts the Service away from being a dependency when testing a Component

## Example of a Service

```
// logger.service.ts
export class Logger {
  log(msg: any) { console.log(msg) };
  error(msg: any) { console.error(msg) };
  warn(msg: any) { console.warn(msg) };
}

// app.component.ts
...
export class AppComponent {
  constructor(private logger: Logger) {
    this.logger.log(`Hello World`);
    this.logger.error(`Hello Error`);
    this.logger.warn(`Hello Warning`);
  }
}
```

An instance of the Logger service is injected into the constructor of the Component

The Logger service's methods are available to component through the injected instance

# Dependency Injection

Wired into the Angular framework

- Used everywhere to provide new components with services (or other things) that they may need

Components CONSUME Services

- Can inject a Service into a component and then use the methods within the Service class

Services usually have the `@Injectable()` decorator

- Provides metadata to allow Angular to inject it into components as a dependency
- Also used to indicate that a component (or other class) HAS a dependency

Injector is an application wide mechanism

- Created during bootstrap process and additionally, as needed – no need to create one
- Creates dependencies and maintains a container of dependency instance to reuse if possible

Provider is an object that tells an injector how to obtain or create a dependency

QA

## Testing Angular Services

**Test set up**

**Synchronous and Asynchronous Testing Services with Dependencies and Angular's TestBed**

**Testing Services with HTTP service calls**



 **Jasmine**

Angular Logo from: <https://angular.io/presskit>

# Testing Services

```
// Testing services with Jasmine
describe(`Testing the service`, () => {
  let serviceToBeTested: TestService;
  let expectedData = [`some`, `data`, `structure`];
  beforeEach(() => { serviceToBeTested = new TestService(); });
  // Synchronous Data
  it(`getAllData should return all of the data`, () => {
    expect(serviceToBeTested.getAllData()).toEqual(expectedData);
  });
  // Asynchronous Data - Functions returning Observables
  it(`getObservableData should return value from observable`,
    (done: DoneFn) => {
      serviceToBeTested.getObservableData().subscribe(value => {
        expect(value).toBe(observableValue);
        done();
      });
    });
});
});
```

Services can be tested without assistance from Angular's testing utilities

Can be written as straight Jasmine test suites

**done()** is generally passed to functions that use asynchronous data  
It signifies that the asynchronous work has finished

## Testing Services with Angular

Sometimes services will be injected into other services and/or components

- Done through Angular's Dependency Injection
- DI will find or create other services too if dependent service has its own dependencies

Service consumers need not worry about this

Services testers must think about first level of service dependency at least

- Angular DI can do service creation
- Done using Angular's **TestBed** test utility

## Angular TestBed

Most important of Angular's testing utilities

- Creates dynamically-constructed Angular test module emulating an Angular `@NgModule`
- `TestBed.configureTestingModule()` method takes `metadata` that can have most of properties of `@NgModule`
- When testing a service, providers in metadata contains array of services needed to test or mock

Instantiation of service can be done in `beforeEach` call or in `it` call

- Up until Angular 8, `TestBed.get(ServiceName)` is used (not type-safe)

```
serviceToBeTested = TestBed.get(TestService) // returns type any
```

- From Angular 9, `TestBed.inject(ServiceName)` is preferred as it is type-safe

```
serviceToBeTested = TestBed.inject(TestService) // returns type TestService
```

Mocking Services is covered in Component Testing

## Using TestBed with Services

```
describe(`SimpleService testing using providers`, () => {
  let service: SimpleService;
  const expectedData = [`some`, `sample`, `data`];

  beforeEach(() => {
    TestBed.configureTestingModule({ providers: [SimpleService] });
    service = TestBed.inject(SimpleService); // get() in < Angular 9
  });

  it('should use the SimpleService', () => {
    service = TestBed.inject(SimpleService); // get() in < Angular 9
    expect(service.getAllData).toEqual(expectedData);
  });
});
```

## **QuickLab 10a – Providing and Using a Service using BDD/TDD – Parts 1-5**



In these parts of the QuickLab, you will:

- Set up a Service
  - Write tests for service functionality
  - Write code for service functionality
- (Up to the end of Part 5)



## Using Services

**Providing a Service**

**Injecting a service into a Component (Dependency Injection)**

**Making service calls from the Component**



Angular Logo from: <https://angular.io/presskit>

# Dependency Injection

Any dependency in an app must be provided

- Register a provider with the app's injector

Injector can use provider to create new instances (if one doesn't currently exist or is not suitable)

For a Service, provider is usually the class itself

When a dependency on a Service is found:

- Angular checks to see if the injector has any existing instances of that Service
- If found, it will return that instance of the Service
- If not found, it will create an instance of the Service using the registered provider and adds it to the injector

## Providing a Service - root

Need to register at least one provider of service to be used

- Can be part of Service's own metadata – making Service available everywhere

```
...
@Injectable({
  providedIn: `root`
})
...
```

- Using Angular CLI to create a Service (`ng g service myService`) registers provider with root injector through the metadata in the `@Injectable`
- Providing a Service at root level means that there is a single, shared instance of the Service
- Can be injected into any class that asks for it
- Optimises app by allowing removal of Service from compiled app if it isn't used

Angular 9 extends the vocabulary for the `providedIn` property to include the platform and any scopes

## Providing a Service - Module

Services can be provided by particular Modules by including them in the providers section of the NgModule metadata

```
...  
@NgModule({  
  ...  
  providers: [MyService]  
})  
...
```

Shorthand for: `providers: [{provide: MyService, useClass: MyService}]`

The **provide** gives Angular a token to use as its internal reference for locating a dependency value and registering the provider

The **useClass** is the provider definition object, a recipe for creating the dependency value.

So you could return something else whenever something asks for **MyService** (such as UserService)

## Providing a Service - Component

Sometimes you may wish to provide a unique instance of a service in a component or module

- Can do this by adding it to the providers array in their configuration objects.  
Beware: this way isn't tree-shakeable and so leads to larger bundle sizes.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [MyService]
})
```

New instance of service is created with each new instance of the Component

## Injecting a Service Injection and Calls in a Component

```
import { Component, OnInit } from '@angular/core';
import { MyService } from './myService.service';
@Component({...})
export class MyComponent implements OnInit {
  private dataFromService: any;
  constructor(private myService: MyService)
  {}
  ngOnInit() {
    this.dataFromService =
      this.myService.myServiceHelperMethod();
  }
}
```

Services have to be injected into Components for them to be used

- Usual to inject an instance of the service into the constructor of the component

Calls to any of the service methods can then be made in any part of the component as required



## Testing Angular Services

Testing Components with Service Dependencies



Angular Logo from: <https://angular.io/presskit>

# Component Class Testing (with Service Dependencies)

```
...
class MockSomeService {
  someServiceMethod = () => { // some code // }
}

...
beforeEach(() => {
  // Provide component and service
  TestBed.configureTestingModule({
    declarations: [UnderTestComponent],
    providers: [
      {
        provide: SomeService,
        useClass: MockSomeService
      }
    ]
  });
})

// Inject component and service
comp = TestBed.inject(UnderTestComponent); // get() in <Angular v9
someService = TestBed.inject(SomeService); // rather than inject()
});

...
it('some spec', () => {
  // some code
  expect(comp.someService.someServiceMethod())
    .toBe('some value');
});
...

```

Components with dependencies can be tested in isolation of them

- Can create an instance of the dependency used in the spec via the **TestBed**
- Can MOCK the dependency in the spec – can make mock meet the minimum needs of the component  
→ Provide and inject both the component and the service in the **TestBed** configuration

Component can then be exercised, using all of its methods (including lifecycle hooks if required)

- Service method calls will defer to the mock
- Spies can be used to check service calls

## **QuickLab 10a – Providing and Using a Service using BDD/TDD – Parts 6-9**



In these parts of the QuickLab, you will:

- Test and implement the use of the service's functions in the application's components

## Services and HTTP

**JSON Server**

**Using HttpClient**

**Importing the module**

**Injecting**

**Requesting Data**

**Sending Data**

**Handling Errors**



Angular Logo from: <https://angular.io/presskit>

## JSON Server

Developing a solution that needs a RESTful API can be troublesome if the service is not available

JSON Server is a "full fake REST API with **zero-coding in less than 30 seconds** (seriously)"

- Essentially needs:

A properly-formed JSON file for the data

An npm installation (either global or local)

Global is recommended by us as you will [probably] use it lots!

```
npm i -g json-server
```

Once installed, point JSON server at your **.json** file

```
json-server --watch /path/to/file/filename.json
```

By default it runs on port 3000 – change this by adding a **-p <PORT\_NO>** to the command

<https://www.npmjs.com/package/json-server>

# JSON Server

```
{  
  "posts": [  
    {"id": 1, "title": "json-server" }  
,  
  "comments": [  
    {"id": 1, "body": "comment", "postID": 1}  
,  
  "profile": {"name": "typicode" }  
}  
  
// So a request to:  
// http://localhost:3000/posts/1  
// will yield:  
{ id: 1, title: json-server }
```

If you have a JSON file as shown, access is by using the format:

`http://localhost:port/arrayName/objectId`

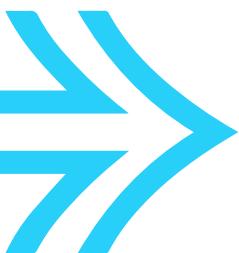
You can use any type of HTTP verb request:

- GET
- POST
- PUT
- UPDATE
- DELETE

The JSON file is modified by requests to do so

- id is immutable and autogenerated

## **QuickLab 10b – Installing JSON Server – Part 1**



In this part of the QuickLab, you will:

- Install JSON server
- Run JSON server using a data file

## Using HttpClient (1)

Most applications use some form of communication with a backend server via HTTP protocol

Services in an application will use HTTP protocol to send and retrieve data needed by the application

**HttpClient** is an Angular module that makes this process simple

It allows:

- Strong typing of request and response objects
- Testability support
- Request and response interceptor support
- Better error handling based on Observables

## Using HttpClient (2) – Importing

`HttpClientModule` needs to be imported into the application so that `HttpClient` can be used

- Most commonly into `app.module.ts`
- Means it only needs to be done once per application
- `HttpClient` can then be injected into any component or service that requires it

```
...
import { HttpClientModule } from '@angular/common/http';
...
@NgModule({
  imports: [
    ...
    HttpClientModule,
    ...
  ],
  ...
})
```

## Using HttpClient (3) – Injecting and Requesting JSON

**HttpClient** is injected into the constructor of the component or service

Requests are made using the **get()** method and subscribing to the returned Observable

```
@Component({...})
export class SomeComponentOrService implements OnInit{

  reqResults: string[];

  constructor(private http: HttpClient) {} //Injecting HttpClient

  ngOnInit(): void {
    this.http.get('my/data/source/url').subscribe(data => {
      this.reqResults = data[someResults]           // Make the HTTP request
      // Read the results
    });
  }
}
```

Type-checking can be done via an interface and using this as a generic on the **get()** method

An error handler can be added to the subscribe call

HttpClient supports the following methods:

request, delete, get, head, jsonp, options, patch, post and put

For more information see: <https://angular.io/api/common/http/HttpClient>

## Note on Service, Components and Observables

```
// Method in Service Class:  
getData(): Observable<Type> {  
  return this.http.get<Type>(URL);  
}  
  
// Method in Component Class  
private data: Type;  
ngOnInit() {  
  this.service.getData().subscribe(  
    data => this.data = data;  
  );  
};
```

It is usual for a method in a Service to return an **Observable**

This means that it needs to be subscribed to at the origin of the request to the Service's method

Type here could be an observable of any type:

- string[]
- number[]
- MyCustomClass[]

## **QuickLab 10b – Getting Data – Parts 2–4**



In these parts of the QuickLab, you will:

- Set up a service to make HTTP calls
- Return data as an Observable from the service
- Subscribe to the service's Observable in the Component

## Using HttpClient (4) – Sending Data

Most commonly done through POST requests to the server

```
const body = // Some data object  
  
http.post('my/data/source/url', body).subscribe()
```

.**subscribe()** is still needed as this initiates the request

Other parts of the request can be configured by providing 3<sup>rd</sup> argument to post method

- Such as Headers and URL parameters
- Uses other classes such as **HttpHeader** and **HttpParams** from the @angular/common/http module
- Uses the set method to pass in the arguments

Again, if a service is making the request it should return an Observable which is subscribed to in the Component accessing the service

## **QuickLab 10b – Sending Data – Parts 5-6**



In these parts of the QuickLab, you will:

- Make a service send some data to a REST API
- Initiate requests in the component by subscribing to the service's Observable

## Using HttpClient (5) – Error Handling

```
this.httpClient.get('my/data/source/url').subscribe(  
  data => {  
    this.reqResults = data[someResults]  
  },  
  (err: HttpErrorResponse) => {  
    if (error instanceof Error) {  
      console.log(`An error occurred:  
        ${err.error.message}`);  
    }  
    else {  
      console.log(`Backend return code:  
        ${err.status}, error was: ${err.error}`);  
    }  
  }  
);
```

Angular provides an **HttpErrorResponse**, imported from `@angular/common/http`

Can be used for error handling as a second argument to `subscribe()`

- The same pattern can be used if the subscription is made to a Service's method in a Component

## **QuickLab 10b – Checking for Errors – Parts 7–8**



In these parts of the QuickLab, you will:

- Make the component handle Observable errors gracefully
- Display errors within the template to advise users



## Testing Service HTTP Calls

**Ensuring a service makes the right calls**

**Setting up the tests**

**Expecting data**

**Checking errors**

**Checking method calls**



Angular Logo from: <https://angular.io/presskit>

## Testing a service with HTTP Service calls

It will be usual for Services to make HTTP calls

- Angular's `HttpClient` service is typically injected and delegated to
- Use a `spy` to test that the Service-under-test makes the right calls to the HTTP service
- Allows Service's methods to be tested independent of the HTTP calls
- Provide a sample of the data expected to use as the expected values from the test

## Test Set Up

```
...
describe(`HttpService call tests`, () => {
  let httpClientSpy: { get: jasmine.Spy };
  let service: TestService;

  beforeEach(() => {
    httpClientSpy = jasmine.createSpyObj(
      `HttpClient`,
      [`get`]
    );
    service = new TestService(
      <any>httpClientSpy
    );
    // test specs to go here
  });
  ...
});
```

In the test suite:

- Define an object for `HttpClient` spy containing the method(s) to be tested
- Define an instance of the service to test
- In `beforeEach`:
  - Create a Spy Object for the `HttpClient` spy
  - Instantiate the service to test, injecting the Spy rather than a real instance of `HttpClient`

## Test Specs that use HttpClient calls

```
...
describe(`HttpClient call tests`, () => {
  // set up goes here
  it(`should return expected data, calling
HttpClient.get once`, () => {
    const expectedData: string = `data`;
    httpClientSpy.get.and.returnValue(asyncData(
      expectedData));
    service.getData().subscribe(
      values => expect(values).toEqual(
        expectedData, `expected data`),
      fail
    );
    expect(httpClientSpy.get).
      toHaveBeenCalledTimes(1);
  });
});
...
```

Define some expected data for the test

Make a call to the method in the Spy

- Return the expected data here too  
→ Note: an additional function **asyncData** has been imported here – it is available from:

<https://github.com/angular/angular/blob/master/aio/content/examples/testing/src/testing/async-observable-helpers.ts>

Subscribe to the Observable returned by the service

- Pass in the values and expect them **toEqual** the expected data or fail

Expect the method of the **HttpClient** spy to have been called exactly once

# HTTP Errors

```
...
describe(`HttpService call tests`, () => {
  // set up goes here
  it(`should return error when server returns error`, () => {
    const errorResponse = new HttpErrorResponse({
      error: `test 404 error`,
      status: 404,
      statusText: `Not Found`
    });
    httpClientSpy.get.and.returnValue(asyncError(
      errorResponse));
    service.getData().subscribe(
      values => fail(`expected an error`),
      error => expect(error.error).
        toContain(`test 404 error`)
    );
  });
});
...
```

A service should deal with Errors returned from `HttpClient` calls

A test specs should be written to check this:

- Define an `HttpErrorResponse` object
- Use a Spy to return an error
- Check that the Observable returns an error when provided with the `HttpResponseError`
  - See if the error contains the text it is provided with

## **QuickLab 10c – Test HttpClient calls – Part 1**



In this part of the QuickLab, you will:

- Test that the service makes the expected calls to its methods and that the methods return the expected data
- Test that the service responds correctly to a 404 server error

QA

## Testing HTTP Requests

Mocking Philosophy

Set Up

Expecting and answering requests

Testing for errors



Jasmine

Angular Logo from: <https://angular.io/presskit>

## Testing HTTP requests – Mocking Philosophy

HTTP backend needs to be mocked so tests can simulate interaction with remote server

`@angular/common/http/testing` library makes mocking straightforward

Designed for pattern of testing when app executes code and makes requests first

- Test expects that certain requests have or have not been made
- Then performs assertions against requests
- Then provides responses by “flushing” each expected request
- Tests may verify that code hasn’t made any unexpected requests

# Test Set Up

```
import {HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';
import { TestBed } from '@angular/core/testing';
import { HttpClient,HttpErrorResponse } from '@angular/common/http';
describe(`HttpClient testing`, () => {
  let httpClient: HttpClient;
  let httpTestingController: HttpTestingController;
  const testUrl = `/data`;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule]
    });
    // Inject the http service and test controller for each test
    httpClient = TestBed.inject(HttpClient); // get() < Angular 9
    httpTestingController = TestBed.inject(
      HttpTestingController
    );
  });
  // Tests begin
});
```

Angular provides an **HttpClientTestingModule** and **HttpTestingController** (a mocking controller)

**HttpClientTestingModule** is added to **TestBed**

Service-under-test is then set up

Tests will use the testing backend instead of the called backend

**HttpClient** service and mocking controller are injected meaning they can be referenced in tests

# Expecting and Answering Requests

```
...  
    // Set up here  
    it(`can test HttpClient.get`, () => {  
      const testData: string = `someData`;  
      // Make an HTTP GET request  
      httpClient.get<string>(testUrl).subscribe(  
        data => expect(data).toBe(testData)  
      );  
      // expectOne() matches the request's URL  
      // None or multiple requests would cause error  
      const req = httpTestingController.expectOne(`/data`);  
      // Assert that method used was GET  
      expect(req.request.method).toEqual(`GET`);  
      // Respond with mock data - Observable resolves  
      req.flush(testData);  
      // Check for no outstanding requests - could be in  
      // afterEach if used throughout suite  
      httpTestingController.verify();  
    });  
});
```

Tests can now be written to make requests and a mock response can be provided

## Testing For Errors

```
it(`test 404 handling`, () => {
  const errmsg = `generated 404 error`;

  httpClient.get<any[]>(testUrl).subscribe(
    data => fail(`fail with 404 error`),
    (error: HttpErrorResponse) => {
      expect(error.status).toEqual(404,
        `status`);
      expect(error.error).toEqual(errmsg,
        `message`);
    }
  );
  const req =
    httpTestingController.expectOne(testUrl);
  req.flush(errmsg, {status: 404, statusText: `Not found`});
});
```

Good practice points to handling HTTP requests that fail

Calling `request.flush()` with an error message could be used to handle a 404 status

## Testing For Errors

```
it(`test for network error`, () => {
  const errmsg = `generated network error`;
  httpClient.get<any[]>(testUrl).subscribe(
    data => fail(`fail with network error`),
    (error: HttpErrorResponse) => {
      expect(error.error.message).toEqual(errmsg,
        `message`);
    }
  );
  const req =
    httpTestingController.expectOne(testUrl);
  // Mock ErrorEvent - for connection timeouts, etc
  const mockError = new ErrorEvent(`Network error`, {
    message: errmsg,
  });
  // Respond with mock error
  req.error(mockError);
}) ;
```

Alternatively use  
`request.error()` with an  
ErrorEvent

## **QuickLab 10c – Mock and Test HttpClient calls – Part 2**



In this part of the QuickLab, you will:

- Mock the HttpClient and check the URL the call was made to against the URL defined in the service – returning a 404 error if it does not match



## Objectives

- To understand when and how to create a Service and provide them in the appropriate place
- To be able to create tests for Services
- To be able to use Services through dependency injection
- To be able to make Services interact with REST APIs using Observables



Angular Logo from: <https://angular.io/presskit>



## **APPENDIX: Services and Observables For Development Purposes**

**Making a Service return an  
observable from static data**



Angular Logo from: <https://angular.io/presskit>

## Static Data

- We have seen that http requests can easily generate an Observable due to their static nature
- Sometimes it may be necessary to use static data as a mock rather than an actual or simulated RESTful service
- In this instance, the of Observable operator should be used in the service to create an Observable from an imported file
- Consider a file that has an array of objects that is imported into the Services
  - The code opposite shows how an Observable can be made from it

```
...
import { Observable, of } from rxjs
import { DATA } from './data'
@Injectable({})
export class DataService {
  ...
  getAllData(): Observable<Data[]> {
    return of(DATA);
  }
}
```



# Routing

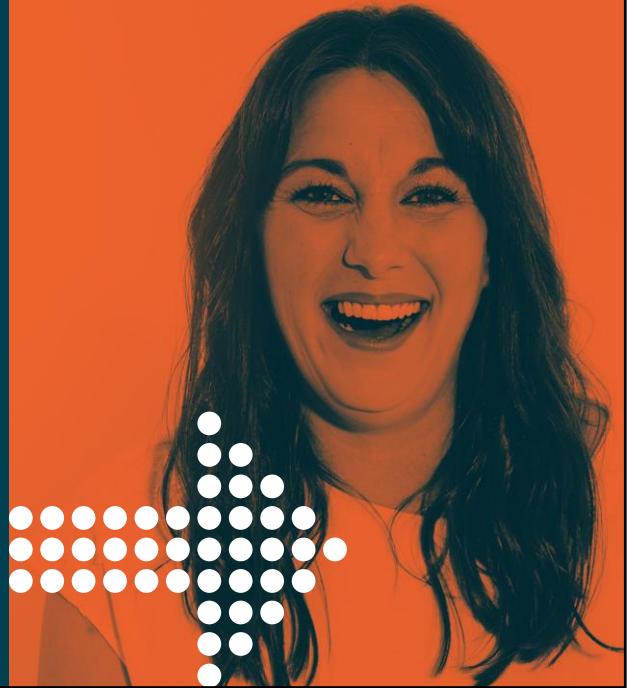
Building Web Applications using Angular





## Objectives

- To be able to create a new application with routing enabled
- To understand how Router Modules are configured
  - Defining arrays of Route objects for specific paths to display specific components
  - Supplying data to a component on a route
  - Redirecting routes
  - Dealing with 404s
- To understand RouteState, ActivatedRoutes and RouterEvents
- To be able to add links to templates for navigation
- To be able to add child routes to an application
- To be able to use Route Guards to control navigation



Angular Logo from: <https://angular.io/presskit>



## Preparing for routing

Using the CLI to generate a project

set up for routing

The RouterModule

The RouterOutlet



# Routing

The mechanism that allows the Angular application to be presented in a series of navigable views

Users expect to be able to navigate an application in the following ways:

- Entering a specific address into the URL to display a particular view/page
- Clicking on links within the page to display another view/page
- Using the browser's back and forward buttons to return and advance to pages/views in the history

Angular provides a Router that can:

- Interpret a browser URL as an instruction to navigate to a particular view
- Pass optional parameters to the view component to help specify the content to display
- Bind links on the template to navigate to the appropriate application view when clicked
- Use other imperative navigation (buttons, selections, etc.) to change the view
- Record in the browser's history so back and forward buttons can be used as expected

## Setting up the application for Routing

Common to add a `<base>` element to **index.html** as first child in `<head>`

- Tells router how to compose navigation URLs from this point
- If **app** folder is in same folder as **index.html** then line below should be added to **index.html**

```
...  
<head>  
  <base href="/">  
  ...  
</head>  
...
```

## Setting up the application for Routing

The Router is an optional service to present specified component for a particular URL

- Not part of Angular Core
- Can be added when setting up a new application when using the CLI

```
:QuickLabs % $ ng new solution  
? Would you like to add Angular routing? (y/N) █
```

- Adds file **app-routing.module.ts** to the applications base files creating **AppRoutingModule**
- Imports **AppRoutingModule** into **AppModule**
- Adds **<router-outlet></router-outlet>** to bottom of **app.component.html**

**NOTE: This can be done manually if this operation is skipped**

# The Router Module

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
// Note: no routes specified yet
const routes: Routes[] = [];
@NgModule({
  imports: [RouterModule.forRoot(
    routes,
  )],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

## Imports

By default, **RouterModule** and **Routes** are imported from the library package **@angular/router**

## Configuration

Routed application has singleton **Router** service

No routes defined until it is configured

**routes** array contains **Route** definitions

Decorator specifies **RouterModule.forRoot** method

- This adds result to **AppModule**'s imports

## Router Outlet

**RouterOutlet** is a directive

- From router library
- Used like a component
- Placeholder to put the template of the Component specified for a route

Places component as a sibling to `<router-outlet></router-outlet>`

Sibling changes as the route changes

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

## **QuickLab 11a – Setting up an application with Routes**



In this QuickLab, you will:

- Set up a project with Routing using the CLI
- Examine the effects of this on the provided code



## Adding Routes

The routesArray

Route

Defining routes



## The routes Array

```
const routes: Routes = [
  {
    path: 'page2',
    component: Page2Component
  },
  {
    path: 'page3',
    component: Page3Component
    data: {key: 'value'}
  }
];
```

Describes how to navigate

- Passed to the `RouterModule.forRoot` method to configure the router
- Each Route supplied maps URL path to a component
- No leading slashes in the `path`
- Router parses and builds final URL
  - Can use both relative and absolute paths when navigating
- Order in routes matters
  - Uses ***first-match wins*** strategy
  - More specific routes should be first

# The Route object

A **Route** can have several optional parameters – some of the most common are:

Route parameter	Description	Options
<b>path</b>	String that uses the route matcher Domain Specific Language	Any string
<b>pathMatch</b>	String that specifies the matching strategy	<code>prefix (default), full</code>
<b>component</b>	Name of component to display when route is activated	Any string
<b>redirectTo</b>	URL fragment which will replace current matched segment	Any string
<b>data</b>	Additional data provided to a component by <b>ActivatedRoute</b>	Any object
<b>outlet</b>	Name of the outlet the component should be placed into	Any string
<b>children</b>	An array of child route definitions	An array of <b>Routes</b>

```
// Create a simple route to a component called MyComponent
{ path: 'mycomponent', component: MyComponent }
```

```
// Create a simple route to a component supplying some data
{ path: 'mycomponent', component: MyComponent, data: {data:
'someValue'}}
```

```
// Redirect a route to the mycomponent route
{ path: 'pathToRedirect', redirectTo: '/mycomponent'}
```

```
// Create a 'catch-all' route to display a '404' page
{ path: '**', component: My404Component }
```



## Testing Routes

**Testing with the RouterTestingModule**

**Writing Specs**



# RouterTestingModule and set Up

```
import { routes } from './app-routing.module';
describe(`Routing test`, () => {
  let location: Location; // from @angular/common
  let router: Router; // from @angular/router
  let fixture: ComponentFixture<AppComponent>
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [RouterTestingModule.withRoutes(routes)],
      declarations: [AppComponent, TestComp1, TestComp2]
    });
    router = TestBed.inject(Router); // get() <v9
    location = TestBed.inject(Location); // get() <v9
    fixture = TestBed.createComponent(AppComponent);
    router.initialNavigation();
  });
  // Tests go here...
});
```

Router and Location need to be accessed during testing

The routes array to be tested needs to be added to the **RouterTestingModule**

Any components used in routes need to be declared

**fixture** is needed so that **router** has **<router-outlet>** to insert components into

Initial navigation performed to set up location change listener

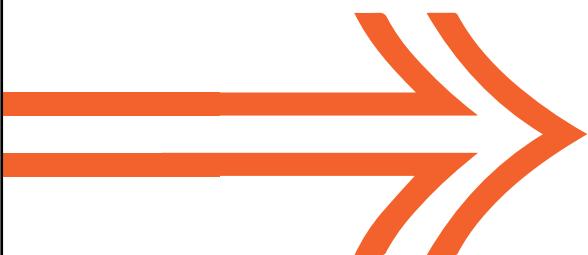
## RouterTestingModule and set Up

```
import { routes } from './app-routing.module';
describe(`Routing test`, () => {
  // Set up here...
  it(`navigate to "" redirects to /testcompl`,
    fakeAsync(() => {
      router.navigate(['']).then(() => {
        expect(location.path()).toBe('/testcompl');
      });
    }));
  // More tests here...
});
```

Tests need to be run using `fakeAsync()`

`router.navigate([''])` returns a **Promise** of a **location**, which, when fulfilled, can be used to compare the returned `location.path()` to the expected location

## **QuickLab 11b – Adding and Testing Routes**



In this QuickLab, you will:

- Generate some components to use in routing
- Create a Routes array to define the path and the component to display
- Test that the routing works as expected

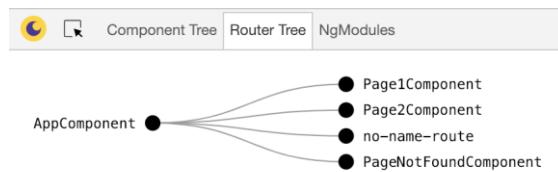
# Router State

Navigation lifecycle triggers router to build a tree of `ActivatedRoute` objects

- Constitutes current state of the router
- Accessing `RouterTree` in Augury shows this

Can access `RouterState` from anywhere in app

- Use `Router` service and `routerState` property



## Activated Routes

Each provides methods to navigate router tree to access information from routes:

- Parent
- Child
- Sibling

**ActivatedRoute** service injected into component to access route path and its parameters

- Information contained here can be seen in the table opposite

Property	Description
<code>url</code>	<code>Observable</code> of route path as an array of strings for each part of path
<code>data</code>	<code>Observable</code> of <code>data</code> object provided for the route
<code>paramMap</code>	<code>Observable</code> map of required and optional parameters specific to route
<code>queryParamMap</code>	<code>Observable</code> map of query parameters available to all routes
<code>fragment</code>	<code>Observable</code> of URL fragment available to all routes
<code>outlet</code>	Name of <code>RouterOutlet</code> used to render route - default is <code>primary</code>
<code>routeConfig</code>	Route configuration used for the route containing origin path
<code>parent</code>	Route's parent <code>ActivatedRoute</code> when route is a child
<code>firstChild</code>	First <code>ActivatedRoute</code> in list of this route's child routes
<code>children</code>	All child routes activated under current route

341

## Router Events

Each navigation triggers the emission of a number of events from the `Router.events` property

The triggering of these can be observed by adding `{enableTracing: true}` as a second argument to the `forRoot` call in the Module's imports – useful for debugging

Event	Description	Event	Description
<code>NavigationStart</code> <code>NavigationEnd</code>	Triggered when navigation starts or ends	<code>ActivationStart</code> <code>ActivationEnd</code>	Triggered when Router begins or finishes activating a route
<code>RouteConfigLoadStart</code> <code>RouteConfigLoadEnd</code>	Triggered before or after Router lazy-loads route config	<code>ResolveStart</code> <code>ResolveEnd</code>	Triggered when Router begins or finishes Resolve phase of routing
<code>RoutesRecognized</code>	Triggered when Router parses URL and routes are recognised	<code>NavigationCancel</code>	Triggered when navigation is cancelled - usually due to a Route Guard returning false
<code>GuardsCheckStart</code> <code>GuardsCheckEnd</code>	Triggered when Router begins or completes Guards phase of routing	<code>NavigationError</code>	Triggered when navigation fails due to an unexpected error
<code>ChildActivationStart</code> <code>ChildActivationEnd</code>	Triggered when Router starts or completes activating a route's children	<code>Scroll</code>	Triggered to represent a scrolling event

`{enableTracing: true}` is added for debugging purposes – it outputs every router event to the browser console during the navigation lifecycle.

## **QuickLab 11c – Using Activated Route to get data**



In this QuickLab, you will:

- Use static data via a route
- Access the ActivatedRoute object to subscribe to its Observable for data
- Use the observed data in the component to set its title



## **Creating links in the template**

**RouterLink**  
**RouterLinkActive**



## RouterLink

Added to the template to allow the user to navigate

Still use `<a>` tags

- Replace attribute `href` with `routerLink`
- Provide additional attribute `routerLinkActive` to specify a CSS class when the route is active

```
<nav>
  <a routerLink="/page1" routerLinkActive="active">Page 1</a>
  <a routerLink="/page2" routerLinkActive="active">Page 2</a>
</nav>
<router-outlet><router-outlet>
```

## RouterLinkActive

Directive that toggles CSS classes for active **RouterLink** bindings based on current **RouterState**

**routerLinkActive** property can be bound directly using a space separated string

```
...  
  <a routerLink="/page1" routerLinkActive="active">Page 1</a>  
...
```

Can also be made to evaluate to a class property by surrounding in square brackets and supplying variable name as value

```
...  
  <a routerLink="/page1" [routerLinkActive]="activeClassProp">Page 1</a>  
...
```

- **activeClassProp** could be an string array containing **classNames** to be applied

## RouterLinkActive

Cascades down through each level of route tree

- Parent and child routes can be active at same time
- Override by binding to `[routerLinkActiveOptions]` with `{exact: true}`  
Given `RouterLink` will only be active if URL is an **exact** match to current URL

```
...
<a
  routerLink="/page1"
  routerLinkActive="active"
  [routerLinkActiveOptions]="{exact: true}"
>
  Page 1
</a>
...
```

## **QuickLab 11d – Adding Links to the Template**



In this QuickLab, you will:

- Add some routes to a template using the routerLink directive
- Style active links using the routerLinkActive directive



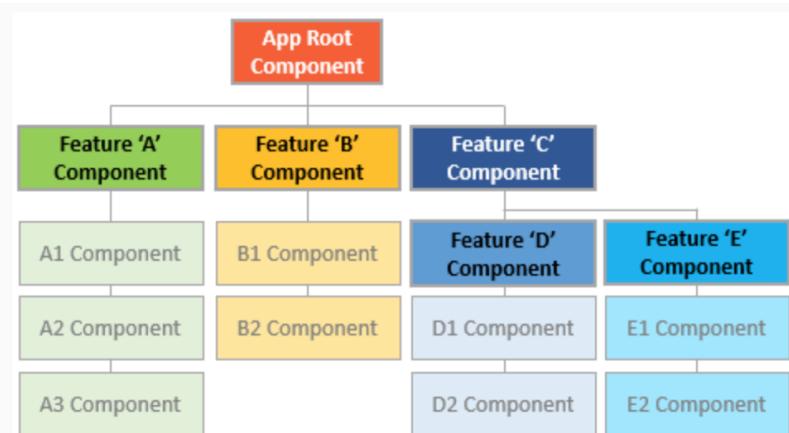
## Child routes

Additional Routing Modules

Child route set up



## Child Routes



Often feature modules need to specify their own routing

In diagram, **Feature C** component would need its own routing

Done by creating its own **routing module**, **routes array** that includes a children array of child routes and using **Router.forChild()**

The CLI can be used to generate a Feature Module with its own routing module:

```
ng g module featurec --route featurec --module app.module
```

## Child Routes – Module Routing

```
// feature-c-routing.module.ts
... //imports
const featureCRoutes: Routes = [
  {
    path: 'featurec',
    component: FeatureCComponent,
    children: [
      {
        path: '',
        component: FeatureCSubComponent
      },
      {
        path: 'featureCDetail',
        component: FeatureCHomeComponent
      }
    ]
];
... // Module decorator and class
```

This allows **Feature C's** routing to be decoupled from the main application as the routing is part of its own module. For **FeatureCComponent** to display its **children** routes it must have its own **<router-outlet>** in its template.

## Child Routes – Module Routing

```
// feature-c-routing.module.ts
... // (imports and routes)
@NgModule({
    imports: [RouterModule.forChild(featureCRoutes)],
    exports: [RouterModule]
})
export class FeatureCRoutingModule {}
```

```
// feature-c.module.ts → import() into AppModule
... // (imports)
@NgModule({
    declarations: [...],
    imports: [
        CommonModule,
        FeatureCRoutingModule
    ]
})
export class FeatureCModule {}
```

`Routes.forChild()` is used to specify that this has routes for children included.

The `RouterModule` must be exported so that this module can be imported into **FeatureC**'s *main module*.

## Making the App aware of Module Routing

Decoupling doesn't mean disconnecting!

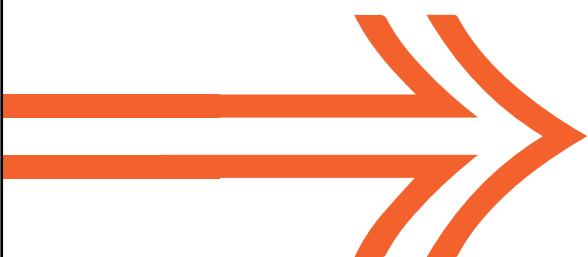
Importing a 'feature' module that has a routing module attached to it makes the whole application aware of the routing it contains

- **GOTCHA** – Any feature module that contains routing *MUST* be BEFORE the app's own routing module in the list of imports in the decorator – otherwise they are not recognised...

Routes can be added to navigation that use the 'feature' modules routing

- If the module is not imported and a 'catch-all' route is supplied in the app's routing this will be activated as if the route does not exist in the application

## QuickLab 11e – Working with Child Routes



In this QuickLab, you will:

- Use feature routing to decouple modules from the main application using `forChild`
- Use child routes within the feature module



## Parameterised Routes

Route Parameters

Getting route parameters

Observable data

Imperative Routing

Optional parameters



## Parameter Routes

For Component reuse with data, sometimes parameters are needed within the URL

These parameters may represent a unique detail about the data to display

- Eg. The id

A parameter is added to the route using a colon after the slash and the name to give the parameter

```
...  
  { path: 'path/to/component/:parameter', component: DetailComponent }  
...
```

The specified component must deal with the Observable to use the data it is supplied from the parameter(s)

## Parameter Routes – Obtaining Data

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, paramMap } from '@angular/router';
import { Observable } from 'rxjs';
import { switchMap } from 'rxjs/operators';
import { DetailsService } from './details.service';
@Component({ ... })
export class DetailComponent implements OnInit {
  details$ = Observable<Details[]>;
  constructor(
    private route: ActivatedRoute,
    private service: DetailsService) {}
  ngOnInit() {
    this.details$ = this.route.paramMap.pipe(
      switchMap((params: paramMap) =>
        this.service.getDetail(params.get('parameter'))
      )
    );
  }
}
```

`ActivatedRoute` can supply an Observable (`ParamMap` or `ParamQueryMap`)

- Can be used by the Component to obtain information from the URL of the current route
- Values from `ParamMap` can be accessed and used as part of arguments to service calls
  - Given that the service that provides data returns an Observable of the data!
  - Often piped into `switchMap`
  - This cancels previous in-flight requests
  - Returns data using new request

From <https://angular.io/guide/rx-library#naming-conventions-for-observables>

Naming conventions for observables

Because Angular applications are mostly written in TypeScript, you will typically know when a variable is an observable. Although the Angular framework does not enforce a naming convention for observables, you will often see observables named with a trailing “\$” sign.

This can be useful when scanning through code and looking for observable values. Also, if you want a property to store the most recent value from an observable, it can be convenient to simply use the same name with or without the “\$”.

## Using the Observable Data

As long as the **Component Template** is set up with the **async** pipe, the data will be displayed when available

```
...  
  <div *ngIf="detail$ | async as detail">  
    <p>{{detail.data}}</p>  
  </div>  
...
```

## Generating Parameterised Routes in the Template

```
<ol>
  <li *ngFor="let detail of details$ | async">
    <a
      [routerLink]=["/path/to/details/", detail.parameter]
      routerLinkActive="active">
      {{detail.data}}
    </a>
  </li>
</ol>
```

Links can be generated dynamically when the Component is initialised

Simply add the parameter name required to the `routerLink` in the template as it is being rendered

Note that `[]` surround the `routerLink` so that the expression is **evaluated** rather than **literal**

## Imperative Routing

As well as using links with `routerLink` and typing the URL in the address bar, navigation can be done imperatively

Inject a `Router` in to the component (i.e. in the constructor) to navigate from, and then add a method to do the navigation

Uses the router's `navigate` method

- Takes same one-item array of *link parameters* (as would be bound to a `routerLink`)
- Wrapping method can be attached to any event on any of the elements in the DOM

```
... (in component.ts)
    goToRoute() {
        this.router.navigate(['/path/to/route/to']);
    }
...
...
```

## Optional Route Parameters

Required route params are passed in the array as subsequent indices

```
this.router.navigate(['/user', user.id, foo, bar])  
//constructs .../user/id/foo/bar
```

Optional route params can be passed as an object

```
this.router.navigate(['/user', user.id, {foo: bar}])  
//constructs .../user/id;foo=bar
```

Note it's not a query string (separated by a ? And & symbols, but matrix URL notation separated by semicolons ;

These parameters can be retrieved from the **ActivatedRoute** object in the same way as required parameters – but don't stop a router URL match from occurring

## **QuickLab 11f – Working with Parameterised Routes**



In this QuickLab, you will:

- Access the ActivatedRoute object in the component to gain access to the Params Observable
- Use parameters in the URL to display specific data
- Create links dynamically in the template



## Named Outlets



## Named Outlets

Named outlets can be used to display multiple routes

Each template can have one unnamed (primary) outlet, but any number of named outlets

```
<router-outlet name="aside-details"></router-outlet>
```

We then need to update our routing module to link up this named outlet to a path and component

```
{
  path: 'details',
  component: 'DetailsComponent',
  outlet: 'aside-details'
}
```

## Named Outlets

To use this in a `routerLink` we need to bind the named outlet to the `RouterLink`

```
<a [routerLink]="[{outlets: {'aside-details': ['details']} }]">Show  
Details</a>
```

We then need to update our routing module to link up this named outlet to a path and component

```
{  
  path: 'details',  
  component: 'DetailsComponent',  
  outlet: 'aside-details'  
}
```

## Named Outlets

Secondary routes persist as you navigate within a component

```
<a [routerLink]="[{outlets: {'aside-details': ['details']} }] ">Show  
Details</a>
```

If you wish to clear the component then you can do so imperatively

```
closeWindow() {  
  this.router.navigate([{outlets: {'chat-window': null}}]);  
}
```



## Route Guards

**Return Values**

**CanActivate**

**CanActivateChild**

**CanDeactivate**



## Route Guards

Route Guards provide a way for us to guard against certain routes being triggered in certain conditions

You may wish to ensure a user is authorised to view that route, or that you do not navigate away from a route without saving changes the user has made

Guards are set to return true or false, respectively allowing or cancelling the navigation

Guards can re-route navigation as well

Most of the time, Guards will not be able to return a synchronous result and so a `Promise<boolean>` or `Observable<boolean>` are acceptable and the router waits for the resolution

# Route Guards

Guards come in several flavours:

Guard Name	Description
<b>CanActivate</b>	to allow/deny navigation to a route
<b>CanActivateChild</b>	to allow/deny navigation to a child route
<b>CanDeactivate</b>	to allow/deny navigation away from a route
<b>Resolve</b>	to perform data retrieval before activating the new route
<b>CanLoad</b>	to allow/deny navigation to a module loaded asynchronously

Multiple guards at every level are permitted

- **CanDeactivate** and **CanActivateChild** are checked first from the deepest child route to the root route
- **CanActivate** is checked next from the root route to the deepest child route
- If the module is loaded asynchronously then **CanLoad** is checked before loading

## Route Guard Return Values

Guard's return value controls router's behaviour:

Return Value	Router Behaviour
<code>true</code>	Navigation process continues
<code>false</code>	Navigation process stops, current page still displayed
<code>UrlTree</code>	Current navigation cancels and new navigation is started to the returned <code>UrlTree</code>

Router can also tell router to navigate to somewhere else

- Effectively cancels current navigation
- When done inside a guard it should return false

## CanActivate

**CanActivate** route guards can be used to restrict access to particular parts of your application

To create a guard we need:

- A class that implements the **CanActivate** interface (and therefore has a **canActivate** method that returns a **Boolean**)

```
@Injectable()  
export class AuthGuardService implements CanActivate {  
  constructor() { }  
  
  canActivate() {  
    console.log('canActivate called');  
    return true;  
  }  
}
```

## CanActivate

To create a guard we need:

- A *reference* to the guard class in the **Routes** array, for the route that we wish to run the guard against

```
{ path: 'admin', canActivate: [AuthGuardService] }
```

## CanActivateChild

`CanActivate` does not check when you switch from one child route to another  
`CanActivateChild` route guards are used to guard changes between child routes  
We create them in the same way as `CanActivate` guards except with the  
`CanActivateChild` interface

```
@Injectable()
export class AuthGuardService implements CanActivateChild {
  constructor() { }

  canActivateChild() {
    console.log('canActivateChild called');
    return true;
  }
}
```

## CanActivateChild

Then add them to your Routes object as you did with **CanActivate** guards

```
{  
  path: 'admin',  
  canActivate: [AuthGuardService],  
  canActivateChild: [AuthGuardService]  
  children: [...]  
}
```

## **QuickLab 11g – Route Guards – CanActivate and CanActivateChild – Parts 1-2**



In these parts of the QuickLab, you will:

- Implement the `CanActivate` and `CanActivateChild` guards in a service
- Link the guards to particular routes in the array

## CanDeactivate

The **CanDeactivate** guard allows us to stop navigation while we check that navigating away from this route is permissible

Often we will want to check with the user whether they want to save their data first - and if so, we will then need to save that data before executing the navigation

Similar to other route guards we need a class to act as the guard, and then we add it to the route within the Routes array

**CanDeactivate** guards often have different implementation for different components and we do not necessarily want our guard to be aware of this implementation, so we set up an interface to abstract this

## CanDeactivate

First we create our interface for a component that has a **CanDeactivate** guard placed on it

```
export interface CanComponentDeactivate {  
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;  
}
```

Then we create the guard which calls upon this abstracted `canDeactivate()` method when required (the guard is passed the instance of the component in question!)

```
@Injectable()  
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {  
  constructor() {}  
  canDeactivate(component: CanComponentDeactivate) {  
    if (!component) { return true }  
    return component.canDeactivate ? component.canDeactivate() : true;  
  }  
}
```

## CanDeactivate

Finally we implement the guard within the component class

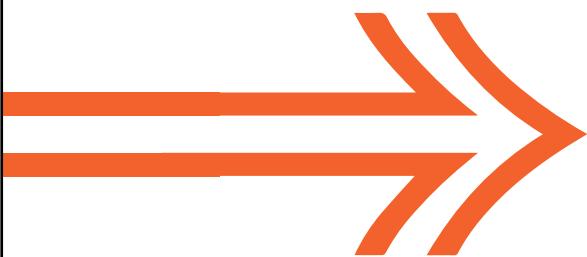
```
export class ManageUsersComponent implements OnInit, CanComponentDeactivate {  
  constructor() { }  
  
  canDeactivate() {  
    return timer(2000)  
  }  
}
```

## CanDeactivate

Not forgetting to declare the guard on the route itself

```
{  
  path: 'users',  
  canDeactivate: [CanDeactivateService],  
  children: [...]  
}
```

## **QuickLab 11g – Route Guards – CanDeactivate – Part 3**



In this part of the QuickLab, you will:

- Create and implement an interface to specify if the component can be navigated away from
- Add the required method to the component
- Attach the guard to the route



## Asynchronous Routing

**Resolve**  
**LazyLoad**  
**CanLoad**



## Resolve

Fetching data asynchronously is a common task, one that will occasionally fail

A Resolve Guard will allow you to attempt to fetch the required data ahead of activating the requested navigation, stopping the attempt if the data request fails

Set up in a very similar way to other guards, we have a guard class which will handle the data request to the service instead of the component itself

# Resolve

```
export class UsersResolver implements Resolve<User[]> {  
  ...  
  resolve(  
    route: ActivatedRouteSnapshot,  
    state: RouterStateSnapshot  
  ): Observable<User[]> {  
    let id = route.params['id'];  
    return this.userService.getUsers().pipe(  
      map(users=>{  
        if (users) {  
          return users;  
        } else {  
          this.router.navigate(['/users']);  
          return null;  
        }  
      })  
    );  
  }  
}
```

The **Resolve** Guard class should implement **Resolve** with the anticipated type of data returned. The guard will then make the request to the service for the data and then handle the success/failure of that request – perhaps redirecting the user to another url should the data not return as expected.

## Resolve

The resolve guard is then added to the route as an object which declares the name of the property where the returned data will be stored

```
resolve: {users: UsersResolver}
```

Which is then retrieved in the component being navigated to

```
this.activatedRoute.data.subscribe((data: {users: User[]}) => {
  this.users = data.users;
})
```

# Lazy Loading

When the user visits our application for the first time they are currently asked to download our entire application. But do they need it? Will they visit every feature? Unlikely.

Lazy loading affords us the ability to earmark certain feature modules for loading only when requested by the user.

To make a module “lazy load” we take two steps:

- Remove all references to it from its parent module (likely to be the root module)
- Add an appropriate path route with `loadChildren` property that gives import instructions for the module

```
{  
  path: 'users',  
  loadChildren: () => import('app/users/users.module')  
    .then(m => m.UsersModule)  
}
```

## CanLoad

So we can use Route Guards to deny loading a particular route, but if that route is lazily loaded we need to be able to stop the asynchronous loading of the requested module – this is the job of the **CanLoad** guard

**CanLoad** works hand-in-hand with **CanActivate** given you want them both to allow/deny a navigation request in unison

```
{  
  path: 'users',  
  loadChildren: () => import('app/users/users.module').then(m => m.UsersModule),  
  canLoad: [AuthGuardService]  
}
```

With appropriate **canLoad()** method

```
canLoad(): boolean {  
  return true;  
}
```

## Key Router Terms and Meanings

Term	Meaning
<b>Router</b>	Displays application component for active URL, managing navigation from one component to the next
<b>RouterModule</b>	Separate <b>NgModule</b> providing necessary service providers and directives for navigating through application views
<b>Routes</b>	Defines an array of <b>Route</b> objects, each mapping URL path to a component
<b>Route</b>	Defines how router navigates to components based on URL pattern – usually path and component
<b>RouterOutlet</b>	Directive ( <code>&lt;router-outlet&gt;</code> ) that specifies where router displays the view
<b>RouterLink</b>	Directive for making clickable HTML elements bind to a route – clicking triggers navigation
<b>RouterLinkActive</b>	Directive for adding/removing classes when route is active/inactive
<b>ActivatedRoute</b>	Provides each route component with route specific information
<b>RouterState</b>	Current state of the router including tree of currently activated routes
<b>Link parameters array</b>	Array interpreted by router as routing instruction to bind to <b>RouterLink</b> or <b>Router.navigate</b>
<b>Routing component</b>	Angular component with a <b>RouterOutlet</b> displaying views based on router navigations



## Objectives

- To be able to create a new application with routing enabled
- To understand how Router Modules are configured
  - Defining arrays of Route objects for specific paths to display specific components
  - Supplying data to a component on a route
  - Redirecting routes
  - Dealing with 404s
- To understand RouteState, ActivatedRoutes and RouterEvents
- To be able to add links to templates for navigation
- To be able to add child routes to an application
- To be able to use Route Guards to control navigation



Angular Logo from: <https://angular.io/presskit>



# Pipes

Building Web Applications using Angular





## Objectives

- To understand what Pipes are and why they are used in Angular
- To become aware of Angular's in-built Pipes
- To be able to create Custom pipes
- To understand the difference between Pure and Impure pipes
- To understand how the AsyncPipe works
- To be able to test Pipes



Angular Logo from: <https://angular.io/presskit>



# Pipes

- In-built Pipes**
- Custom Pipes**
- Pure Pipes**
- Impure Pipes**
- The AsyncPipe**



## Why Pipes?

Applications usually have the purpose to get data, transform it and then output it – often to users

Pushing raw data to views is often not great for user experience

- In financial applications monetary data often stored as integer values  
    Don't want to display prices in the lowest denomination of the currency though!
- Dates stored as Epoch numbers are not user friendly!

Converting our application's data into presentable information is often thought of as a stylistic task

- Well suited to be done in the view

Pipes allow transformation of raw data that can be used in HTML

# Pipes

A pipe takes input data and outputs it in the desired format

```
<p>My name is {{fullname | titlecase}}</p>
```

Angular has a number of built in pipes

AsyncPipe	CurrencyPipe	DatePipe
DecimalPipe	I18nPluralPipe	I18nSelectPipe
JsonPipe	LowerCasePipe	PercentPipe
SlicePipe	TitleCasePipe	UpperCasePipe

## Using Pipes

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<p>Today's date is {{ today | date }}</p>`
})
export class TodaysDateComponent {
  today = new Date();
  // would be 2019-02-28T15:05:16.195Z
}
```

Today's date is Feb 28, 2019

Pipes take an input and output it in the format required

This example shows transforming a raw Date into the format required for display in the template

## Parameterising Pipes

Some pipes can accept any number of optional parameters to help format its output

- Done by adding a colon after the pipe's name and then adding the parameter value
- Values are dependent on the pipe that is being used

Date pipes can be formatted using a string to represent format eg.

```
<p>Date: {{today | date: "MM/dd/yy"}}</p>  <!-- outputs Date: 02/28/19 -->
```

Currency can be formatted using a string to represent which eg.

```
<p>Cost: {{price | currency: "EUR"}}</p>  <!-- outputs Cost: €2.99 -->
```

Multiple values are separated by another colon

```
<p>Cost: {{value | currency: "EUR": "Euros "}}</p>
<!-- outputs Cost: Euros 2.99 -->
```

The slice pipe accepts a start and stop

```
<p>Sliced Word: {{shells | slice:1:5}}</p>
<!-- outputs Sliced Word: hell -->
```

## Using Pipes

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<p>
    Today's date is {{ today | date | uppercase}}
  </p>`
})
export class TodaysDateComponent {
  today = new Date();
}
```

Today's date is FEB 28, 2019

Several pipes can be chained together to create data in the required format  
Simply add another | and then the pipe name (and any parameters)

## Custom Pipes

Custom pipes can be built by creating a pipe class that implements the `PipeTransform` interface and is decorated by the `@Pipe()` decorator with a 'name' property

The class should take 1 + n arguments – the initial value of the data, followed by n arguments that the pipe can receive returning the transformed data

```
@Pipe({
  name: 'custom'
})
export class CustomPipe implements PipeTransform {
  transform(value: string, replace: string, replacement?: string): string {
    if (!replacement) {
      replacement = "bananas"
    }

    return value.replace(replace, replacement);
  }
}
```

## The PipeTransform Interface

`transform` method Essential to a pipe

**PipeTransform** interface defines that method and guides both tooling and compiler

- Technically optional – Angular looks for and executes the transform method regardless

## Notes for using Custom Pipes

Custom Pipes can be used in same way as built-in pipes

Pipe must be included in declarations array of module it is to be used in (or root module)

- Angular reports an error if pipe is not registered with a Module
- Using the CLI to generate a pipe automatically registers it

```
ng g pipe pipe-name
```

## Pure Pipes

Pipes are either **PURE** (by default) or impure

Pure pipes are only executed when Angular detects a pure change to the input value

- A change to a primitive input value or a changed object reference
- Changes in composite objects are ignored

Eg. changes to an input month, data added to an input array, updates in input object property not recognised

May seem restrictive but is fast

Object reference check is faster than deep value check

- Angular quickly determines if pipe execution and view update can be skipped

## Impure Pipes

Executed by Angular during every change detection cycle

- Called as often as necessary eg. on a key-stroke or mouse-move
- Be cautious

Expensive, long-running pipes can destroy user experience

Create by setting pure property in decorator to false

```
@Pipe({
  name: 'custom',
  pure: false
})
export class CustomPipe implements PipeTransform {...}
```

# AsyncPipe

Built-in **IMPURE** pipe

Accepts a Promise or Observable and unwraps emitted values as they arrive

**AsyncPipe** is also stateful

- Maintains a subscription to the input Observable
- Keeps delivering values from the Observable as they arrive

```
 {{ timer | async | date: medium }}
```

```
timer: Observable<Date> = new Observable<Date>(observer => {
  setInterval(()=>{
    observer.next(new Date());
  }, 1000);
})
```

The date is: May 3, 2017, 6:05:41 PM



# Testing Custom Pipes

**Writing tests for pure custom pipes**  
**Testing pipes affect DOM output**  
**Testing pipes with asynchronous data**



## Testing Pipes

```
describe(`UpperCasePipe test`, () => {
  let pipe = new UpperCasePipe();
  it(`transforms abc to ABC`, () => {
    expect(pipe.transform(`abc`)).toBe(`ABC`);
  });
  it(`transforms abc def to ABC DEF`, () => {
    expect(pipe.transform(`abc def`))
      .toBe(`ABC DEF`);
  });
});
```

Easy to test without any Angular testing utilities

Need to simply test that the transform method produces the correct output given an input

Imagine a pipe that converts a word given to it into uppercase

- Test would be as shown

## Testing Pipe Outputs in the DOM

```
describe(`TestComponent`, () => {
  let fixture: ComponentFixture<TestComponent>;
  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [TestComponent, TestPipe]
    }).compileComponents();
    fixture = TestBed.createComponent(TestComponent);
  }));
  it(`should output words in capital letters`, () => {
    const hostEl = fixture.nativeElement;
    const pipedEl = hostEl.querySelector('p');
    expect(pipedEl.textContent).toBe(`CAPITALS`);
  });
});
```

The same as testing a DOM output from a component

Expect content of the element the pipe output is rendered in to match the expected output

Angular's TestBed is required here to obtain the DOM element and access its output

# Testing Asynchronous Pipe Outputs in the DOM

```
describe(`TestComponent`, () => {
  let fixture: ComponentFixture<TestComponent>;
  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [TestComponent, TestPipe]
    }).compileComponents();
    fixture = TestBed.createComponent(TestComponent);
  }));
  it(`should output words in caps`, async() => {
    const hostEl = fixture.nativeElement;
    const pipedEl = hostEl.querySelector('p');
    fixture.whenStable().then(() => {
      expect(pipedEl.textContent).toBe(`CAPITALS`);
    });
  });
});
```

Similar to testing a synchronous pipe  
It spec should be  
async

Wait for the fixture to  
become stable before  
running the tests  
using the  
whenStable() testing  
method

## QuickLab 12 – Pipes



In this QuickLab, you will:

- Experiment with the date and async built-in pipes
- Create a custom pipe to separate words
- If you have time, write some tests for the custom pipe



## Objectives

- To understand what Pipes are and why they are used in Angular
- To become aware of Angular's in-built Pipes
- To be able to create Custom pipes
- To understand the difference between Pure and Impure pipes
- To understand how the AsyncPipe works
- To be able to test Pipes



Angular Logo from: <https://angular.io/presskit>



## Course Outcomes

By the end of the course, you will be able to:

- Explain what the building blocks of Angular are
- Create a Single Page Application that:
  - Uses a number of components with data binding
  - Uses custom directives and pipes
  - Leverages the power of RxJS Observables
  - Uses Services to handle data
  - Is fully routed
  - Is tested





**Hope you enjoyed this  
training course.**





**Please complete your  
evaluation**

**Code: QAANGULAR**

**PIN: 4652093-29**

