

Angular

2/4/5

Name | Course Title | Date

# FLOW

Practices:

- https://angular.io/guide/styleguide

- https://github.com/rangle/angular2-guidelines

Angular:

- Building Blocks

: Module [ES2015 Module, Angular Module]

: Component

: Template

: Metadata (Decorator)

: Data Binding

: Directives

: Service

: Dependency Injection (Injector)

- Angular Bootstraping [bootstrapModule(root module)]

- Angular compilation

: Optimises the angular application logic for the target environment

like, browser or mobiles etc.

- types : AOT (static), Dynamic (on-demand)

- Angular Modules (@NgModule)

: Properties

- declarations : list of members

- imports : list of dependencies (modules)

- bootstrap : root component (tobe instantiated at the start up)

- exports : configured only in library modules, to expose members to

dependant modules.

- providers : list of services to be managed at module level

: Root module (instantiated to begin the angular application execution)

- Component

: @Component

- Properties

: selector - custom element for component

: template - inline html

: templateUrl - external html template file

: styles : inline css

: styleUrls : external css files

: animations : component animation definitions

: providers : list of services to be injected to components

- Interaction

: @Input - Parent to Child

- Use [Property of child] property binding to pass data to child

: @Output - Child to Parent

- Should use with "EventEmitter" type

- Hooks

: ngOnChanges, ngOnInit, ngAfterContentInit, ngAfterContentChecked

, ngAfterViewInit, ngAfterViewChecked, ngOnDestroy

- Data Binding

: {{expression}} - interpolation, Component to DOM

: [dom property] - property binding, Component to DOM

: (angular event) - event binding, DOM to Component

- Service

: @Injectable

- Observable vs. Promise

: Deferred execution, means execute on subscription

: Delay the execution on subscribe [debouncing]

: Terminate the current execution

: Can trigger events/callback many times

: would offer event notification facility through special types

to monitor the context based ont the data like,

- Subject

- BehaviorSubject

- SPA [Single Page Applications]

: Use Router Module for Navigation - @angular/router

: Configure Valid URLs as Route Table using RouterModule

: Setup

- Models

1. ng g class models\cart-item

2. ng g class models\user

- Components

1. ng g component home

2. ng g component authLogin

3. ng g component cartInfo

// admin module

4. ng g component admin\users

5. ng g component admin\reports

- Services

1. ng g service services/auth

2. ng g service services/cart

- Module

1. ng g module appRouter (for routing)

2. ng g module admin

- install underscorejs library

npm i --save underscore

-Form (@angular/forms)

: Types

- Template driven (declarative)

: ngForm, ngModel

- Model Driven or Reactive forms (imperative)

: FormBuilder, FormGroup, FormControl

: ngForm/ ngModel

- valid, invalid, dirty, pristine, errors, touched, untouched

- pseudo classes: ng-pristine, ng-valid, ng-invalid, ng-dirty

: Validation

: 2-way binding

- Routing

: "Router" service for initiating navation through code

: Guards

- to execute logic before navigation

- canActivate, canDeActivate, canChildActivate, canChildDeActivate,

canLoad, canResolve

: Lazy Loadin of modules

- configure the nested routes with module using "loadChildren"

- Pipe

: Which acts on data before binding

: Ment for manipulating the data not DOM

: Built-in

- currency, uppercase, lowercase, date, number

- async

: Custom

- @Pipe, Interface - PipeTransform

: types

- Statelesss, Stateful

- Directives

: Used in templates to define binding logic and achieve DOM manipulation

: ngIf, ngFor, ngSwitch, ngClass, ngModel, ngForm

: Types

- Template Directives or Structural directives

: For custom directives use,

TemplateRef (for getting copy of the child template content)

and ViewContainerRef (for accessing parent DOM object)

- Attribute directives

- i18n & Internalization

1.

2. Extract the i18n marked text using Angular i18n tools

ng-xi18n -p tsconfig.json --i18nFormat=xlf / xlf-2 / xmb

- Ploymer2 with Ng2:

Source: https://codeload.github.com/hotforfeature/origami/zip/master

Github: vaadin

- Angular 4

: ES2015 flattened moudle support

: Universal

- Server side rendering

: ExpressJS app, Ng4 Server modules

- SEO

: Dynamic Components

: \*nIf - ngElse

``````````````````````````````````````````````````````````

Angular CLI:

To manage angular projects.

Setup Angular CLI:

- npm i -g @angular-cli

To Create Project

- ng new Ng2Ex

To Create components

- ng g component path\name

To Create models or types

- ng g class path\name

To run unit tests

ng test

ng test --code-coverage (with coverage reort)

To run e2e tests

ng e2e

To build app

ng build --dev / --prod

To unplug ng from project

ng eject --prod

```````````````````````````````````````````````````

TypeScript:

Install Typescript

npm i -g typescript

- OOP, Classes, Interface, Inheritance

- Modules (export/import)

- Programmable String

- Short Hand Syntax ( Lambda expressions )

: Functions

: Assignments

- Scopes

: Local (function scope), Global (window/tab)

: Block Scope ( if{}, for(){} )

if() {

let x = 10;

var y = 20;

}

------------------------------------------------------

Setup RESTful service:

- Install Json-server

npm i -g json-server

- Start server

json-server --watch --port 9090 catalog.json

------------------------------------------------------

Downloads:

<http://muralisetty.com/trng/T/NG2ExTmpls.zip>

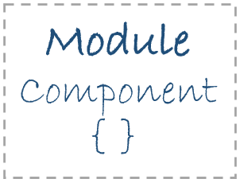
# Introduction

* Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript.
* The framework consists of several libraries, some of them core and some optional.
* You write Angular applications by:
  + composing HTML templates with Angularized markup
  + writing component classes to manage those templates
  + adding application logic in services
  + boxing components and services in modules
* Then you launch the app by bootstrapping the root module.



# Building Blocks

## Modules



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

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

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

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

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

Sample:

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

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

@NgModule({

imports: [ BrowserModule ],

providers: [ Logger ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Launch an application by bootstrapping its root module. During development you're likely to bootstrap the AppModule in a main.ts file like this one.

import { enableProdMode } from '@angular/core';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule);

## NgModules vs. JavaScript modules

The NgModule — a class decorated with @NgModule — is a fundamental feature of Angular.

* In JavaScript each file is a module and all objects defined in the file belong to that module.
* The module declares some objects to be public by marking them with the export key word.
* Other JavaScript modules use import statements to access public objects from other modules.

The NgModule classes differ from JavaScript module in the following key ways:

* An NgModule bounds [declarable classes](https://angular.io/guide/ngmodule-faq#q-declarable) only. Declarables are the only classes that matter to the [Angular compiler](https://angular.io/guide/ngmodule-faq#q-angular-compiler).
* Instead of defining all member classes in one giant file as in a JavaScript module, you list the module's classes in the @[NgModule.declarations](https://angular.io/api/core/NgModule" \l "declarations) list.
* An NgModule can only export the [declarable classes](https://angular.io/guide/ngmodule-faq#q-declarable) it owns or imports from other modules. It doesn't declare or export any other kind of class.
* Unlike JavaScript modules, an NgModule can extend the entire application with services by adding providers to the @[NgModule.providers](https://angular.io/api/core/NgModule" \l "providers) list.

Angular modularity

Modules are a great way to organize an application and extend it with capabilities from external libraries.

NgModules consolidate components, directives, and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow, or common collection of utilities.

Feature Modules

With feature modules, you can keep code related to a specific functionality or feature separate from other code. Delineating areas of your app helps with collaboration between developers and teams, separating directives, and managing the size of the root module.

Angular libraries



Angular ships as a collection of JavaScript modules. You can think of them as library modules.

Each Angular library name begins with the @angular prefix.

You install them with the npm package manager and import parts of them with JavaScript import statements.

For example, import Angular's Component decorator from the @angular/core library like this:

import { [Component](https://angular.io/api/core/Component) } from '@angular/core';

In the example of the simple root module above, the application module needs material from within that BrowserModule. To access that material, add it to the @NgModule metadata imports like this.

imports: [ [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) ],

## Components

* A component controls a patch of screen called a view.
* Components are the most basic building block of an UI in an Angular application. An Angular application is a tree of Angular components.
* Angular components are a subset of directives. Unlike directives, components always have a template and only one component can be instantiated per an element in a template.

A component must belong to an NgModule in order for it to be usable by another component or application. To specify that a component is a member of an NgModule, you should list it in the declarations field of that NgModule.

In addition to the metadata configuration specified via the Component decorator, components can control their runtime behavior by implementing various Life-Cycle hooks.

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

export class HeroListComponent implements [OnInit](https://angular.io/api/core/OnInit) {

heroes: Hero[];

selectedHero: Hero;

constructor(private service: HeroService) { }

ngOnInit() {

this.heroes = this.service.getHeroes();

}

selectHero(hero: Hero)

{

this.selectedHero = hero;

}

}

Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle through optional lifecycle hooks, like ngOnInit() declared above.

@Component Decorator Properties

* outputs - list of class property names that expose output events that others can subscribe to
* providers - list of providers available to this component and its children
* selector - css selector that identifies this component in a template
* styleUrls - list of urls to stylesheets to be applied to this component's view
* styles - inline-defined styles to be applied to this component's view
* template - inline-defined template for the view
* templateUrl - url to an external file containing a template for the view
* viewProviders - list of providers available to this component and its view children
* animations - list of animations of this component

More about component decorator: <https://angular.io/api/core/Component>

### @Input

@Input decorator binds a property within one component (child component) to receive a value from another component (parent component). This is one way communication from parent to child. The component property should be annotated with @Input decorator to act as input property. A component can receive a value from another component using component property binding. To use alias for the binding property name we need to assign an alias name as @Input(alias).

Find the use of @Input with string data type.

@Input()

ctMsg : string;

@Input('ctArray')

myctArray : Array<string>

@Input('stdLeader')

myStdLeader : Student;

### @Output

@Output decorator binds a property of a component to send data from one component (child component) to calling component (parent component). This is one way communication from child to parent component. @Output binds a property of the type of angular EventEmitter class. This property name becomes custom event name for calling component. @Output decorator can also alias the property name as @Output(alias) and now this alias name will be used in custom event binding in calling component.   
Find the @Output decorator using aliasing.

@Output('addStudentEvent')

addStdEvent = new EventEmitter<Student>();

### Custom Event Binding using @Output and EventEmitter

Here we will discuss custom event binding using @Output decorator step by step.   
1. Create text box using element property binding in child component. input event is fired when there is any change in text box. $event.target.value fetches the current value of text box entered by user.

<div>

First Number :<input (input)="num1=$event.target.value" /> <br/>

Second Number:<input (input)="num2=$event.target.value" /> <br/>

<br/> <button (click)="addNumber()">Add Number</button>

</div>

2. Find the method created in child component that will be fired when click event is invoked on click of button from above (step-1) code snippet. emit() is the method of EventEmitter class that emits event payload.

addNumber() {

this.addNumEvent.emit(parseInt(this.num1) + parseInt(this.num2));

}

3. In the child component, create an instance of EventEmitter annotated by @Output decorator. This instance will work as custom event name. Here we are using aliasing for custom event name.

@Output('addNumberEvent')

addNumEvent = new EventEmitter<number>();

4. Now we are performing custom event binding. The custom event addNumberEvent will be invoked in parent component when emit() method is invoked from child component. The event payload is accessed by $event object.

<child-two (addNumberEvent) = "printSum($event)" > </child-two>

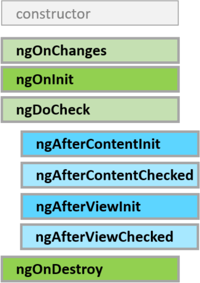
5. The event addNumberEvent will call printSum() method.

printSum(res) {

this.sum = res;

}

## Lifecycle Hooks



A component has a lifecycle managed by Angular.

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

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

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

export class PeekABoo implements OnInit {

constructor(private logger: LoggerService) { }

// implement OnInit's `ngOnInit` method

ngOnInit() { this.logIt(`OnInit`); }

logIt(msg: string) {

this.logger.log(`#${nextId++} ${msg}`);

}

}

## **Lifecycle sequence**

After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

|  |  |
| --- | --- |
| **Hook** | **Purpose and Timing** |
| ngOnChanges() | Respond when Angular (re)sets data-bound input properties. The method receives a [SimpleChanges](https://angular.io/api/core/SimpleChanges) object of current and previous property values.  Called before ngOnInit() and whenever one or more data-bound input properties change. |
| ngOnInit() | Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.  Called once, after the first ngOnChanges(). |
| ngDoCheck() | Detect and act upon changes that Angular can't or won't detect on its own.  Called during every change detection run, immediately after ngOnChanges()and ngOnInit(). |
| ngAfterContentInit() | Respond after Angular projects external content into the component's view.  Called once after the first ngDoCheck().  A component-only hook. |
| ngAfterContentChecked() | Respond after Angular checks the content projected into the component.  Called after the ngAfterContentInit() and every subsequent ngDoCheck().  A component-only hook. |
| ngAfterViewInit() | Respond after Angular initializes the component's views and child views.  Called once after the first ngAfterContentChecked().  A component-only hook. |
| ngAfterViewChecked() | Respond after Angular checks the component's views and child views.  Called after the ngAfterViewInit and every subsequent ngAfterContentChecked().  A component-only hook. |
| ngOnDestroy() | Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.  Called just before Angular destroys the directive/component. |

## Templates

You define a component's view with its companion template. A template is a form of HTML that tells Angular how to render the component.

A template looks like regular HTML, except for a few differences. Here is a template for our HeroListComponent.

<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>

<ul>

<li \*ngFor="let hero of heroes" (click)="selectHero(hero)">

{{hero.name}}

</li>

</ul>

<app-hero-detail \*ngIf="selectedHero" [hero]="selectedHero"></app-hero-detail>

## Metadata

Metadata tells Angular how to process a class.

In TypeScript, you attach metadata by using a decorator. Here's some metadata for HeroListComponent:

@[Component](https://angular.io/api/core/Component)({

selector: 'app-hero-list',

templateUrl: './hero-list.component.html',

providers: [ HeroService ]

})

export class HeroListComponent implements [OnInit](https://angular.io/api/core/OnInit) { /\* . . . \*/ }

## DATA BINDING

Angular supports data binding, a mechanism for coordinating parts of a template with parts of a component. Add binding markup to the template HTML to tell Angular how to connect both sides.



The HeroListComponent [example](https://angular.io/guide/architecture#templates) template has three forms:

src/app/hero-list.component.html (binding)

<li>{{hero.name}}</li>

<app-hero-detail [hero]="selectedHero"></app-hero-detail>

<li (click)="selectHero(hero)"></li>

* The {{hero.name}} [*interpolation*](https://angular.io/guide/displaying-data#interpolation) displays the component's hero.name property value within the <li>element.
* The [hero] [*property binding*](https://angular.io/guide/template-syntax#property-binding) passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.
* The (click) [*event binding*](https://angular.io/guide/user-input#click) calls the component's selectHero method when the user clicks a hero's name.

Two-way data binding is an important fourth form that combines property and event binding in a single notation, using the [ngModel](https://angular.io/api/forms/NgModel) directive. Here's an example from the HeroDetailComponent template:

src/app/hero-detail.component.html (ngModel)

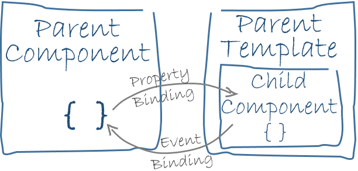
<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name">

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

Angular processes *all* data bindings once per JavaScript event cycle, from the root of the application component tree through all child components.



Data binding plays an important role in communication between a template and its component.



Data binding is also important for communication between parent and child components.

INTERPOLATION

You use interpolation to weave calculated strings into the text between HTML element tags and within attribute assignments. {{1 + 1}}

The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property. {{title}}

The expression can invoke methods of the host component. {{1 + 1 + getVal()}}

|  |  |  |
| --- | --- | --- |
| **Data direction** | **Syntax** | **Type** |
| One-way from data source to view target | {{expression}}  [target]="expression"  bind-target="expression" | Interpolation Property Attribute Class Style |
| One-way from view target to data source | (target)="statement"  on-target="statement" | Event |
| Two-way | [(target)]="expression"  bindon-target="expression" | Two-way |

### **Binding targets**

|  |  |  |
| --- | --- | --- |
| **Type** | **Target** | **Examples** |
| Property | Element property Component property Directive property | src/app/app.component.html  <img [src]="heroImageUrl">  <app-hero-detail [hero]="currentHero"></app-hero-detail>  <div [[ngClass](https://angular.io/api/common/NgClass)]="{'special': isSpecial}"></div> |
| Event | Element event Component event Directive event | src/app/app.component.html  <button (click)="onSave()">Save</button>  <app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>  <div (myClick)="clicked=$event" clickable>click me</div> |
| Two-way | Event and property | src/app/app.component.html  <input [([ngModel](https://angular.io/api/forms/NgModel))]="name"> |
| Attribute | Attribute (the exception) | src/app/app.component.html  <button [attr.aria-label]="help">help</button> |
| Class | class property | src/app/app.component.html  <div [class.special]="isSpecial">Special</div> |
| Style | style property | src/app/app.component.html  <button [style.color]="isSpecial ? 'red' : 'green'"> |

## DIRECTIVES

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

A directive is a class with a @[Directive](https://angular.io/api/core/Directive) decorator. A component is a *directive-with-a-template*; a @[Component](https://angular.io/api/core/Component) decorator is actually a @[Directive](https://angular.io/api/core/Directive) decorator extended with template-oriented features.

While a component is technically a directive, components are so distinctive and central to Angular applications that this architectural overview separates components from directives.

Two *other* kinds of directives exist:

1. *Structural* directives
2. *Attribute* directives

They tend to appear within an element tag as attributes do, sometimes by name but more often as the target of an assignment or a binding.

Structural directives alter layout by adding, removing, and replacing elements in DOM.

The [example template](https://angular.io/guide/architecture#templates) uses two built-in structural directives:

src/app/hero-list.component.html (structural)

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes"></li>

<app-hero-detail \*[ngIf](https://angular.io/api/common/NgIf)="selectedHero"></app-hero-detail>

* [\*ngFor](https://angular.io/guide/displaying-data#ngFor) tells Angular to stamp out one <li> per hero in the heroes list.
* [\*ngIf](https://angular.io/guide/displaying-data#ngIf) includes the HeroDetail component only if a selected hero exists.

Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

The [ngModel](https://angular.io/api/forms/NgModel) directive, which implements two-way data binding, is an example of an attribute directive. [ngModel](https://angular.io/api/forms/NgModel) modifies the behavior of an existing element (typically an <input>) by setting its display value property and responding to change events.

src/app/hero-detail.component.html (ngModel)

<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name">

Angular has a few more directives that either alter the layout structure (for example, [ngSwitch](https://angular.io/guide/template-syntax" \l "ngSwitch)) or modify aspects of DOM elements and components (for example, [ngStyle](https://angular.io/guide/template-syntax" \l "ngStyle) and [ngClass](https://angular.io/guide/template-syntax" \l "ngClass)).

Of course, you can also write your own directives. Components such as HeroListComponent are one kind of custom directive.

## Services

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

Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

here is nothing specifically *Angular* about services. Angular has no definition of a service. There is no service base class, and no place to register a service.

Yet services are fundamental to any Angular application. Components are big consumers of services.

Here's an example of a service class that logs to the browser console:

src/app/logger.service.ts (class)

export class Logger {

log(msg: any) { console.log(msg); }

error(msg: any) { console.error(msg); }

warn(msg: any) { console.warn(msg); }

}

Here's a HeroService that uses a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) to fetch heroes. The HeroService depends on the Logger service and another BackendService that handles the server communication grunt work.

src/app/hero.service.ts (class)

export class HeroService {

private heroes: Hero[] = [];

constructor(

private backend: BackendService,

private logger: Logger) { }

getHeroes() {

this.backend.getAll(Hero).then( (heroes: Hero[]) => {

this.logger.log(`Fetched ${heroes.length} heroes.`);

this.heroes.push(...heroes); // fill cache

});

return this.heroes;

}

}

@Injectable() services

Notice that the new service imports the Angular Injectable symbol and annotates the class with the @Injectable() decorator.

The @Injectable() decorator tells Angular that this service might itself have injected dependencies. It doesn't have dependencies now but it will soon. Whether it does or it doesn't, it's good practice to keep the decorator.

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

@Injectable()

export class MessageService {

messages: string[] = [];

add(message: string) {

this.messages.push(message);

}

clear() {

this.messages = [];

}

}

## dEPENDENCY iNJECTION

Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

An injector maintains a container of service instances that it has previously created. If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular. When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments. This is dependency injection.

The process of HeroService injection looks a bit like this:



If the injector doesn't have a HeroService, how does it know how to make one?

In brief, you must have previously registered a provider of the HeroService with the injector. A provider is something that can create or return a service, typically the service class itself.

You can register providers in modules or in components.

Points to remember about dependency injection:

* Dependency injection is wired into the Angular framework and used everywhere.
* The *injector* is the main mechanism.
  + An injector maintains a *container* of service instances that it created.
  + An injector can create a new service instance from a *provider*.
* A *provider* is a recipe for creating a service.
* Register *providers* with injectors.

## Other building blocks

[Animations](https://angular.io/guide/animations): Animate component behavior without deep knowledge of animation techniques or CSS with Angular's animation library.

Change detection: The change detection will cover how Angular decides that a component property value has changed, when to update the screen, and how it uses zones to intercept asynchronous activity and run its change detection strategies.

Events: The events will cover how to use components and services to raise events with mechanisms for publishing and subscribing to events.

[Forms](https://angular.io/guide/forms): Support complex data entry scenarios with HTML-based validation and dirty checking.

[HTTP](https://angular.io/guide/http): Communicate with a server to get data, save data, and invoke server-side actions with an HTTP client.

[Lifecycle hooks](https://angular.io/guide/lifecycle-hooks): Tap into key moments in the lifetime of a component, from its creation to its destruction, by implementing the lifecycle hook interfaces.

[Pipes](https://angular.io/guide/pipes): Use pipes in your templates to improve the user experience by transforming values for display. Consider this currency pipe expression:

price | currency:'USD':true

It displays a price of 42.33 as $42.33.

[Router](https://angular.io/guide/router): Navigate from page to page within the client application and never leave the browser.

[Testing](https://angular.io/guide/testing): Run unit tests on your application parts as they interact with the Angular framework using the *Angular Testing Platform*.

# Angular Bootstrapping

Every application has at least one Angular module, the *root* module that you bootstrap to launch the application. By convention, it is usually called AppModule.

An NgModule describes how the application parts fit together. Every application has at least one Angular module, the *root* module that you bootstrap to launch the application. By convention, it is usually called AppModule.

If you use the CLI to generate an app, the default AppModule is as follows:

/\* JavaScript imports \*/

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';

import { [FormsModule](https://angular.io/api/forms/FormsModule) } from '@angular/forms';

import { [HttpModule](https://angular.io/api/http/HttpModule) } from '@angular/http';

import { AppComponent } from './app.component';

/\* the AppModule class with the @[NgModule](https://angular.io/api/core/NgModule) decorator \*/

@[NgModule](https://angular.io/api/core/NgModule)({

declarations: [

AppComponent

],

imports: [

[BrowserModule](https://angular.io/api/platform-browser/BrowserModule),

[FormsModule](https://angular.io/api/forms/FormsModule),

[HttpModule](https://angular.io/api/http/HttpModule)

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

The @[NgModule](https://angular.io/api/core/NgModule) decorator identifies AppModule as an [NgModule](https://angular.io/api/core/NgModule) class. @[NgModule](https://angular.io/api/core/NgModule) takes a metadata object that tells Angular how to compile and launch the application.

* *declarations*—this application's lone component.
* *imports*—import [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) to have browser specific services such as DOM rendering, sanitization, and location.
* *providers*—the service providers.
* *bootstrap*—the *root* component that Angular creates and inserts into the index.html host web page.

The default CLI application only has one component, AppComponent, so it is in both the declarations and the bootstrap arrays.

## **The imports array**

The module's imports array appears exclusively in the @[NgModule](https://angular.io/api/core/NgModule) metadata object. It tells Angular about other NgModules that this particular module needs to function properly.

This list of modules are those that export components, directives, or pipes that the component templates in this module reference. In this case, the component is AppComponent, which references components, directives, or pipes in [BrowserModule](https://angular.io/api/platform-browser/BrowserModule), [FormsModule](https://angular.io/api/forms/FormsModule), or [HttpModule](https://angular.io/api/http/HttpModule). A component template can reference another component, directive, or pipe when the referenced class is declared in this module or the class was imported from another module.

You don't have any services to provide yet. But you will create some before long and you may choose to provide many of them here.

## **The providers array**

The providers array is where you list the services the app needs. When you list services here, they are available app-wide. You can scope them when using feature modules and lazy loading.

## **The bootstrap array**

The application launches by bootstrapping the root AppModule, which is also referred to as an entryComponent. Among other things, the bootstrapping process creates the component(s) listed in the bootstrap array and inserts each one into the browser DOM.

Each bootstrapped component is the base of its own tree of components. Inserting a bootstrapped component usually triggers a cascade of component creations that fill out that tree.

While you can put more than one component tree on a host web page, most applications have only one component tree and bootstrap a single root component.

This one root component is usually called AppComponent and is in the root module's bootstrap array.

# Angular compilation

An Angular application consists largely of components and their HTML templates. Before the browser can render the application, the components and templates must be converted to executable JavaScript by an *Angular compiler*.

Angular offers two ways to compile your application:

1. *Just-in-Time* (JIT), which compiles your app in the browser at runtime
2. *Ahead-of-Time* (AOT), which compiles your app at build time.

JIT - Compile TypeScript just in time for executing it.

* Compiled in the browser.
* Each file compiled separately.
* No need to build after changing your code and before reloading the browser page.
* Suitable for local development.

AOT - Compile TypeScript during build phase.

The Angular Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads and runs that code.

* Compiled by the machine itself, via the command line (Faster).
* All code compiled together, inlining HTML/CSS in the scripts.
* No need to deploy the compiler (Half of Angular size).
* More secure, original source not disclosed.
* Suitable for production builds.

AoT runs the compile step during the build process instead of inside the browser, so when you build your project for production, the compiler can disappear completely from the output, saving precious CPU cycles when the page loads in the user’s browser.

If you are using angular2, the probability of you using TypeScript (TS) along with angular2 is very high. Since majority of the features of TS are still not supported by the browser, we might be relying on tools such as TypeScript compiler. These convert our code from TS to JS first, and then we serve these JS files to the browser (which again compiles the JS files to a binary which they understand).

Essentially we are compiling the code twice with angular2 apps, once when we convert TS to JS and then when the browser converts JS to binary.

While we cannot control the latter, we can however control when the compilation from TS to JS is performed.

With angular2, if you go with JIT (which is default), both the compiles happen after the code is loaded in the browser (i.e. TS -> JS -> binary). Not only is it an additional overhead to do the TS -> JS compilation on the fly on the browser, but also, the angular2 compiler is almost half the size of the angular2 package so if we avoid this, we can reduce the size of the payload significantly.

Refer: <https://scotch.io/@kashyapmukkamala/jit-vs-aot-in-angular2-and-how-to-use-it>

## **How AOT works**

It helps to think of the AOT compiler as having two phases: a code analysis phase in which it simply records a representation of the source; and a code generation phase in which the compiler's StaticReflector handles the interpretation as well as places restrictions on what it interprets.

## Phase 1: analysis

The TypeScript compiler does some of the analytic work of the first phase. It emits the .d.ts type definition files with type information that the AOT compiler needs to generate application code.

At the same time, the AOT collector analyzes the metadata recorded in the Angular decorators and outputs metadata information in .metadata.json files, one per .d.ts file.

You can think of .metadata.json as a diagram of the overall structure of a decorator's metadata, represented as an abstract syntax tree (AST).

## Phase 2: code generation

The collector makes no attempt to understand the metadata that it collects and outputs to .metadata.json. It represents the metadata as best it can and records errors when it detects a metadata syntax violation.

It's the compiler's job to interpret the .metadata.json in the code generation phase.

## **Observable vs. Promise**

**Observable vs. Promise**

**: Deferred execution, means execute on subscription**

**: Delay the execution on subscribe [debouncing]**

**: Terminate the current execution**

**: Can trigger events/callback many times**

**: would offer event notification facility through special types**

**to monitor the context based on the data like,**

**- Subject**

**- BehaviorSubject**

**Promise**

A Promise handles a **single event** when an async operation completes or fails.

Note: There are Promise libraries out there that support cancellation, but ES6 Promise doesn't so far.

A promise can be:

* **fulfilled** - The action relating to the promise succeeded
* **rejected** - The action relating to the promise failed
* **pending** - Hasn't fulfilled or rejected yet
* **settled** - Has fulfilled or rejected

**Queuing asynchronous actions**

You can also chain thens to run async actions in sequence.

When you return something from a then() callback, it's a bit magic. If you return a value, the next then() is called with that value. However, if you return something promise-like, the next then() waits on it, and is only called when that promise settles (succeeds/fails). For example:

getJSON('story.json').then(function(story) {  
  return getJSON(story.chapterUrls[0]);  
}).then(function(chapter1) {  
  console.log("Got chapter 1!", chapter1);  
})

**Observable**

<https://angular.io/guide/observables>

* Observables provide support for passing messages between publishers and subscribers in your application
* Observables offer significant benefits over other techniques for event handling, asynchronous programming, and handling multiple values
* As a publisher, you create an Observable instance that defines a *subscriber* function. This is the function that is executed when a consumer calls the subscribe() method.
* The subscriber function defines how to obtain or generate values or messages to be published.

: Deferred execution, means execute on subscription

: Delay the execution on subscribe [debouncing]

: Terminate the current execution

: Can trigger events/callback many times

: would offer event notification facility through special types

to monitor the context based ont the data like,

- Subject

- BehaviorSubject

// This function runs when subscribe() is called

function sequenceSubscriber(observer) {

// synchronously deliver 1, 2, and 3, then complete

observer.next(1);

observer.next(2);

observer.next(3);

observer.complete();

// unsubscribe function doesn't need to do anything in this

// because values are delivered synchronously

return {unsubscribe() {}};

}

// Create a new Observable that will deliver the above sequence

const sequence = new Observable(sequenceSubscriber);

// execute the Observable and print the result of each notification

sequence.subscribe({

next(num) { console.log(num); },

complete() { console.log('Finished sequence'); }

});

// Logs:

// 1

// 2

// 3

// Finished sequence

Observables compared to other techniques

Observables are often compared to promises. Here are some key differences:

* *Observables are declarative; computation does not start until subscription*. Promises execute immediately on creation. This makes observables useful for defining recipes that can be run whenever you need the result.
* *Observables provide many values*. Promises provide one. This makes observables useful for getting multiple values over time.
* *Observables differentiate between chaining and subscription*. Promises only have .then() clauses. This makes observables useful for creating complex transformation recipes to be used by other part of the system, without causing the work to be executed.
* Observables subscribe() is responsible for handling errors. Promises push errors to the child promises. This makes observables useful for centralized and predictable error handling.

### **Declarative**

* Observables are not executed until a consumer subcribes. The subscribe() executes the defined behavior once, and it can be called again. Each subscription has its own computation. Resubscription causes recomputation of values.

// declare [a](https://angular.io/api/router/RouterLinkWithHref) publishing operation

new Observable((observer) => { subscriber\_fn });

// initiate execution

observable.subscribe(() => {

// observer handles notifications

});

* Promises execute immediately, and just once. The computation of the result is initiated when the promise is created. There is no way to restart work. All then clauses (subscriptions) share the same computation.

// initiate execution

new Promise((resolve, reject) => { executer\_fn });

// handle return value

promise.then((value) => {

// handle result here

});

### **Chaining**

* Observables differentiate between transformation function such as a map and subscription. Only subscription activates the subscriber function to start computing the values.

observable.map((v) => 2\*v);

* Promises do not differentiate between the last .then clauses (equivalent to subscription) and intermediate .then clauses (equivalent to map).

promise.then((v) => 2\*v);

### **Cancellation**

* Observable subscriptions are cancellable. Unsubscribing removes the listener from receiving further values, and notifies the subscriber function to cancel work.

const sub = obs.subscribe(...);

sub.unsubscribe();

* Promises are not cancellable.

### **Error handling**

* Observable execution errors are delivered to the subscriber's error handler, and the subscriber automatically unsubscribes from the observable.

obs.subscribe(() => {

throw Error('my error');

});

* Promises push errors to the child promises.

promise.then(() => {

throw Error('my error');

});

Real example in project:

//create a subject

private navigateDaysFilter = new Subject<any>();

//create a method that will call the changes

naviagteDaysfilterModal(condition) {

this.navigateDaysFilter.next(condition);

}

//create an Observable

navigateDaysStream$ = this.navigateDaysFilter.asObservable();

//Subscribe to observable

this.daysSubscription = navigation.navigateDaysStream$.subscribe(

navigateDays => {

//code goes here

});

Routing & Navigation

The Angular [Router](https://angular.io/api/router/Router) enables navigation from one [view](https://angular.io/guide/glossary#view) to the next as users perform application tasks.

An introduction to a few core router concepts will help orient you to the details that follow.

### [**link**](https://angular.io/guide/router#base-href)**<base href>**

Most routing applications should add a <base> element to the index.html as the first child in the <head> tag to tell the router how to compose navigation URLs.

If the app folder is the application root, as it is for the sample application, set the href value exactly as shown here.

src/index.html (base-href)

<base href="/">

### **Router imports**

src/app/app.module.ts (import)

import { [RouterModule](https://angular.io/api/router/RouterModule), [Routes](https://angular.io/api/router/Routes) } from '@angular/router';

### **Configuration**

A routed Angular application has one singleton instance of the [*Router*](https://angular.io/api/router/Router) service. When the browser's URL changes, that router looks for a corresponding [Route](https://angular.io/api/router/Route) from which it can determine the component to display.

A router has no routes until you configure it. The following example creates four route definitions, configures the router via the RouterModule.forRoot method, and adds the result to the AppModule's imports array.

src/app/app.module.ts (excerpt)

const appRoutes: [Routes](https://angular.io/api/router/Routes) = [

{ path: 'crisis-center', component: CrisisListComponent },

{ path: 'hero/:id', component: HeroDetailComponent },

{

path: 'heroes',

component: HeroListComponent,

data: { title: 'Heroes List' }

},

{ path: '',

redirectTo: '/heroes',

pathMatch: 'full'

},

{ path: '\*\*', component: PageNotFoundComponent }

];

@[NgModule](https://angular.io/api/core/NgModule)({

imports: [

RouterModule.forRoot(

appRoutes,

{ enableTracing: true } // <-- debugging purposes only

)

// other imports here

],

...

})

export class AppModule { }

### **Router outlet**

Given this configuration, when the browser URL for this application becomes /heroes, the router matches that URL to the route path /heroes and displays the HeroListComponent after a [RouterOutlet](https://angular.io/api/router/RouterOutlet) that you've placed in the host view's HTML.

**<router-outlet></router-outlet>**

**<!-- Routed views go here -->**

### **Router links**

Now you have routes configured and a place to render them, but how do you navigate? The URL could arrive directly from the browser address bar. But most of the time you navigate as a result of some user action such as the click of an anchor tag.

Consider the following template:

**template: `**

**<h1>Angular Router</h1>**

**<nav>**

**<a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>**

**<a routerLink="/heroes" routerLinkActive="active">Heroes</a>**

**</nav>**

**<router-outlet></router-outlet>**

**`**

## **The sample application**

src/app/app.module.ts (first-config)

**import { NgModule } from '@angular/core';**

**import { BrowserModule } from '@angular/platform-browser';**

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

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

**import { AppComponent } from './app.component';**

**import { CrisisListComponent } from './crisis-list.component';**

**import { HeroListComponent } from './hero-list.component';**

**const appRoutes: Routes = [**

**{ path: 'crisis-center', component: CrisisListComponent },**

**{ path: 'heroes', component: HeroListComponent },**

**];**

**@NgModule({**

**imports: [**

**BrowserModule,**

**FormsModule,**

**RouterModule.forRoot(**

**appRoutes,**

**{ enableTracing: true } // <-- debugging purposes only**

**)**

**],**

**declarations: [**

**AppComponent,**

**HeroListComponent,**

**CrisisListComponent,**

**],**

**bootstrap: [ AppComponent ]**

**})**

**export class AppModule { }**

src/app/app.component.ts (template)

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

**@Component({**

**selector: 'app-root',**

**template: `**

**<h1>Angular Router</h1>**

**<nav>**

**<a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>**

**<a routerLink="/heroes" routerLinkActive="active">Heroes</a>**

**</nav>**

**<router-outlet></router-outlet>**

**`**

**})**

**export class AppComponent { }**

### **Wildcard route**

{ path: '\*\*', component: PageNotFoundComponent }

### **feature routing requirements**

### **Hero feature route configuration**

Create a new heroes-routing.module.ts in the heroes folder using the same techniques you learned while creating the AppRoutingModule.

src/app/heroes/heroes-routing.module.ts

**import { NgModule } from '@angular/core';**

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

**import { HeroListComponent } from './hero-list.component';**

**import { HeroDetailComponent } from './hero-detail.component';**

**const heroesRoutes: Routes = [**

**{ path: 'heroes', component: HeroListComponent },**

**{ path: 'hero/:id', component: HeroDetailComponent }**

**];**

**@NgModule({**

**imports: [**

**RouterModule.forChild(heroesRoutes)**

**],**

**exports: [**

**RouterModule**

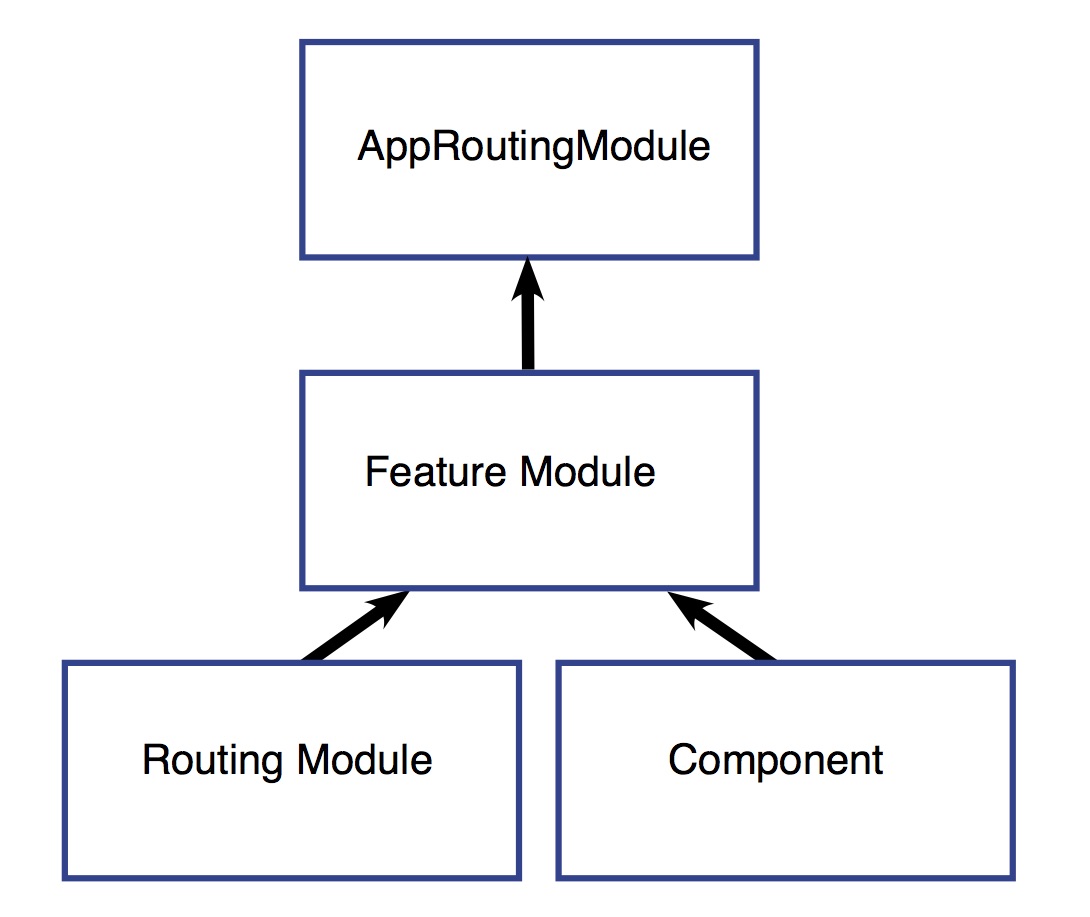
**]**

**})**

**export class HeroRoutingModule { }**

Lazy Loading Feature Modules

The two feature modules, OrdersModule and CustomersModule, have to be wired up to the AppRoutingModule so the router knows about them. The structure is as follows:



### **Routes at the app level**

In AppRoutingModule, update the routes array with the following:

src/app/app-routing.module.ts

const routes: [Routes](https://angular.io/api/router/Routes) = [

{

path: 'customers',

loadChildren: 'app/customers/customers.module#CustomersModule'

},

{

path: 'orders',

loadChildren: 'app/orders/orders.module#OrdersModule'

},

{

path: '',

redirectTo: '',

pathMatch: 'full'

}

];

### **Configure the feature module’s routes**

The next step is in customers-routing.module.ts. First, import the component at the top of the file with the other JavaScript import statements. Then, add the route to CustomerListComponent.

src/app/customers/customers-routing.module.ts

import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';

import { [Routes](https://angular.io/api/router/Routes), [RouterModule](https://angular.io/api/router/RouterModule) } from '@angular/router';

import { CustomerListComponent } from './customer-list/customer-list.component';

const routes: [Routes](https://angular.io/api/router/Routes) = [

{

path: '',

component: CustomerListComponent

}

];

@[NgModule](https://angular.io/api/core/NgModule)({

imports: [RouterModule.forChild(routes)],

exports: [[RouterModule](https://angular.io/api/router/RouterModule)]

})

export class CustomersRoutingModule { }

Angular Modules (Advanced)

Entry Components

An entry component is any component that Angular loads imperatively, (which means you’re not referencing it in the template), by type.

There are two main kinds of entry components:

* The bootstrapped root component.
* A component you specify in a route definition.

## **A bootstrapped entry component**

The following is an example of specifying a bootstrapped component, AppComponent, in a basic app.module.ts:

@[NgModule](https://angular.io/api/core/NgModule)({

declarations: [

AppComponent

],

imports: [

[BrowserModule](https://angular.io/api/platform-browser/BrowserModule),

[FormsModule](https://angular.io/api/forms/FormsModule),

[HttpModule](https://angular.io/api/http/HttpModule),

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent] // bootstrapped entry component

})

A bootstrapped component is an entry component that Angular loads into the DOM during the bootstrap process (application launch). Other entry components are loaded dynamically by other means, such as with the router.

## **A routed entry component**

The second kind of entry component occurs in a route definition like this:

const routes: [Routes](https://angular.io/api/router/Routes) = [

{

path: '',

component: CustomerListComponent

}

];

A route definition refers to a component by its type with component: CustomerListComponent.

All router components must be entry components. Because this would require you to add the component in two places (router and entryComponents) the Compiler is smart enough to recognize that this is a router definition and automatically add the router component into entryComponents.

### **entryComponents and the compiler**

* The code should contain only the classes that you actually need and exclude components that are never used.
* For this reason, the Angular compiler only generates code for components which are reachable from the entryComponents;
* This means that adding more references to @[NgModule.declarations](https://angular.io/api/core/NgModule" \l "declarations) does not imply that they will necessarily be included in the final bundle.
* If a component isn't an entry component and isn't found in a template, the ***Tree Shaker*** will throw it away. So, it's best to add only the components that are truly entry components to help keep your app as trim as possible.

Feature Modules

Feature modules are NgModules for the purpose of organizing code.

## **Feature modules vs. root modules**

While you can do everything within the root module, feature modules help you partition the app into focused areas.

A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.

## **How to make a feature module**

ng generate module CustomerDashboard

Providers

## **Provider scope**

* When you add a service provider to the providers array of the root module, it’s available throughout the app.
* when you import a module that has providers, those providers are also available to all the classes in the app as long they have the lookup token.

## **Limiting provider scope by lazy loading modules**

Angular uses an injector system to make things available between modules. In an eagerly loaded app, the root application injector makes all of the providers in all of the modules available throughout the app.

When the Angular router lazy-loads a module*, it creates a new injector.* Any component created within a lazy loaded module’s context, such as by router navigation, gets the local instance of the service, not the instance in the root application injector. Components in external modules continue to receive the instance created for the application root.

Though you can provide services by lazy loading modules, not all services can be lazy loaded. For instance, some modules only work in the root module, such as the Router. The Router works with the global location object in the browser.

## **Limiting provider scope with components**

Another way to limit provider scope is by adding the service you want to limit to the component’s providers array. Component providers and NgModule providers are independent of each other. This method is helpful for when you want to eagerly load a module that needs a service all to itself. Providing a service in the component limits the service only to that component (other components in the same module can’t access it.)

Singleton services

## **Providing a singleton service**

An injector created from a module definition will have services which are singletons with respect to that injector.

To control the lifetime of services, one controls the creation and destruction of injectors.

For example, a route will have an associated module. When the route is activated, an injector is created from that module as a child of the current injector. When you navigate away from the route, the injector is destroyed. This means that services declared in a route module will have a lifetime equal to that of the route. Similarly, services provided in an application module will have the same lifetime of the application, hence singleton.

forRoot()

If a module provides both providers and declarations (components, directives, pipes) then loading it in a child injector such as a route, would duplicate the provider instances.

The duplication of providers would cause issues as they would shadow the root instances, which are probably meant to be singletons.

For this reason Angular provides a way to separate providers out of the module so that same module can be imported into the root module with providers and child modules without providers.

1. Create a static method forRoot() (by convention) on the module.
2. Place the providers into the forRoot method as follows.

If the [RouterModule](https://angular.io/api/router/RouterModule) didn’t have forRoot() then each route component would instantiate a new [Router](https://angular.io/api/router/Router) instance, which would break the application as there can only be one [Router](https://angular.io/api/router/Router)

## **Prevent reimport of the CoreModule**

Only the root AppModule should import the CoreModule. If a lazy-loaded module imports it too, the app can generate [multiple instances](https://angular.io/guide/ngmodule-faq#q-why-bad) of a service.

To guard against a lazy-loaded module re-importing CoreModule, add the following CoreModule constructor.

src/app/core/core.module.ts

constructor (@[Optional](https://angular.io/api/core/Optional)() @[SkipSelf](https://angular.io/api/core/SkipSelf)() parentModule: CoreModule) {

if (parentModule) {

throw new Error(

'CoreModule is already loaded. Import it in the AppModule only');

}

}

The constructor tells Angular to inject the CoreModule into itself. The injection would be circular if Angular looked for CoreModule in the current injector. The @[SkipSelf](https://angular.io/api/core/SkipSelf) decorator means "look for CoreModule in an ancestor injector, above me in the injector hierarchy."

Here's forRoot() that takes a UserServiceConfig object:

src/app/core/core.module.ts (forRoot)

[static](https://angular.io/api/upgrade/static) forRoot(config: UserServiceConfig): [ModuleWithProviders](https://angular.io/api/core/ModuleWithProviders) {

return {

ngModule: CoreModule,

providers: [

{provide: UserServiceConfig, useValue: config }

]

};

}

NgModule API

At a high level, NgModules are a way to organize Angular apps and they accomplish this through the metadata in the @[NgModule](https://angular.io/api/core/NgModule) decorator. The metadata falls into three categories:

* **Static:** Compiler configuration which tells the compiler about directive selectors and where in templates the directives should be applied through selector matching. This is configured via the declarations array.
* **Runtime:** Injector configuration via the providers array.
* **Composability/Grouping:** Bringing NgModules together and making them available via the imports and exports arrays.

@[NgModule](https://angular.io/api/core/NgModule)({

// Static, that is compiler configuration

declarations: [], // Configure the selectors

entryComponents: [], // Generate the host factory

// Runtime, or injector configuration

providers: [], // Runtime injector configuration

// Composability / Grouping

imports: [], // composing NgModules together

exports: [] // making NgModules available to other parts of the app

})

|  |  |
| --- | --- |
| declarations | A list of [declarable](https://angular.io/guide/ngmodule-faq#q-declarable) classes, (components, directives, and pipes) that belong to this module. |
| providers | A list of dependency-injection providers.  Angular registers these providers with the NgModule's injector. If it is the NgModule used for bootstrapping then it is the root injector. |
| imports | A list of modules which should be folded into this module. Folded means it is as if all the imported NgModule's exported properties were declared here. |
| exports | A list of declarations—component, directive, and pipe classes—that an importing module can use. |
| bootstrap | A list of components that are automatically bootstrapped.  Usually there's only one component in this list, the root component of the application. |
| entryComponents | A list of components that can be dynamically loaded into the view.  By default, an Angular app always has at least one entry component, the root component, AppComponent. Its purpose is to serve as a point of entry into the app, that is, you bootstrap it to launch the app. |

NgModule FAQs

## **What is a declarable?**

Declarables are the class types—components, directives, and pipes—that you can add to a module's declarations list. They're the only classes that you can add to declarations.

## **What does "Can't bind to 'x' since it isn't a known property of 'y'" mean?**

This error often means that you haven't declared the directive "x" or haven't imported the NgModule to which "x" belongs.

Perhaps you declared "x" in an application sub-module but forgot to export it. The "x" class isn't visible to other modules until you add it to the exports list.

## **What should I import?**

Import NgModules whose public (exported) [declarable classes](https://angular.io/guide/bootstrapping#the-declarations-array) you need to reference in this module's component templates.

This always means importing [CommonModule](https://angular.io/api/common/CommonModule) from @angular/common for access to the Angular directives such as [NgIf](https://angular.io/api/common/NgIf) and NgFor. You can import it directly or from another NgModule that [re-exports](https://angular.io/guide/ngmodule-faq#q-reexport) it.

Import [FormsModule](https://angular.io/api/forms/FormsModule) from @angular/forms if your components have [([ngModel](https://angular.io/api/forms/NgModel))] two-way binding expressions.

Import shared and feature modules when this module's components incorporate their components, directives, and pipes.

Import only [BrowserModule](https://angular.io/guide/ngmodule-faq" \l "q-browser-vs-common-module) in the root AppModule.

## **Should I import**[**BrowserModule**](https://angular.io/api/platform-browser/BrowserModule)**or**[**CommonModule**](https://angular.io/api/common/CommonModule)**?**

The root application module, AppModule, of almost every browser application should import [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) from @angular/platform-browser.

[BrowserModule](https://angular.io/api/platform-browser/BrowserModule) provides services that are essential to launch and run a browser app.

[BrowserModule](https://angular.io/api/platform-browser/BrowserModule) also re-exports [CommonModule](https://angular.io/api/common/CommonModule) from @angular/common, which means that components in the AppModule module also have access to the Angular directives every app needs, such as [NgIf](https://angular.io/api/common/NgIf) and NgFor.

Do not import [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) in any other module. Feature modules and lazy-loaded modules should import [CommonModule](https://angular.io/api/common/CommonModule) instead. They need the common directives. They don't need to re-install the app-wide providers.

Importing [CommonModule](https://angular.io/api/common/CommonModule) also frees feature modules for use on any target platform, not just browsers.

## **What is the forRoot() method?**

The forRoot() static method is a convention that makes it easy for developers to configure the module's providers.

The [RouterModule.forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) method is a good example. Apps pass a [Routes](https://angular.io/api/router/Routes) object to [RouterModule.forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) in order to configure the app-wide [Router](https://angular.io/api/router/Router) service with routes. [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) returns a [ModuleWithProviders](https://angular.io/api/core/ModuleWithProviders). You add that result to the imports list of the root AppModule.

Only call and import a .forRoot() result in the root application module, AppModule. Importing it in any other module, particularly in a lazy-loaded module, is contrary to the intent and will likely produce a runtime error. For more information, see [Singleton Services](https://angular.io/guide/singleton-services).

[RouterModule](https://angular.io/api/router/RouterModule) also offers a forChild static method for configuring the routes of lazy-loaded modules.

forRoot() and forChild() are conventional names for methods that configure services in root and feature modules respectively.

Angular doesn't recognize these names but Angular developers do. Follow this convention when you write similar modules with configurable service providers.

## **Why is a service provided in a feature module visible everywhere?**

Providers listed in the @[NgModule.providers](https://angular.io/api/core/NgModule" \l "providers) of a bootstrapped module have application scope. Adding a service provider to @[NgModule.providers](https://angular.io/api/core/NgModule" \l "providers) effectively publishes the service to the entire application.

When you import an NgModule, Angular adds the module's service providers (the contents of its providers list) to the application root injector.

This makes the provider visible to every class in the application that knows the provider's lookup token, or knows its name.

## **What if two modules provide the same service?**

When two imported modules, loaded at the same time, list a provider with the same token, the second module's provider "wins". That's because both providers are added to the same injector.

When Angular looks to inject a service for that token, it creates and delivers the instance created by the second provider.

The service provided by the root AppModule takes precedence over services provided by imported NgModules. The AppModule always wins.

## **How do I restrict service scope to a module?**

When a module is loaded at application launch, its @[NgModule.providers](https://angular.io/api/core/NgModule" \l "providers) have application-wide scope; that is, they are available for injection throughout the application.

Load the module lazily if you can. Angular gives a [lazy-loaded module](https://angular.io/guide/ngmodule-faq#q-lazy-loaded-module-provider-visibility) its own child injector. The module's providers are visible only within the component tree created with this injector.

If you must load the module eagerly, when the application starts, provide the service in a component instead.

## **Should I add application-wide providers to the root AppModule or the root AppComponent?**

Register application-wide providers in the root AppModule, not in the AppComponent.

Lazy-loaded modules and their components can inject AppModule services; they can't inject AppComponent services.

## **Why is it bad if a shared module provides a service to a lazy-loaded module?**

### **The eagerly loaded scenario**

When an eagerly loaded module provides a service, for example a UserService, that service is available application-wide.

If the root module provides UserService and imports another module that provides the same UserService, Angular registers one of them in the root app injector (see [What if I import the same module twice?](https://angular.io/guide/ngmodule-faq#q-reimport)).

Then, when some component injects UserService, Angular finds it in the app root injector, and delivers the app-wide singleton service. No problem.

### **The lazy loaded scenario**

Now consider a lazy loaded module that also provides a service called UserService.

When the router lazy loads a module, it creates a child injector and registers the UserService provider with that child injector. The child injector is not the root injector.

When Angular creates a lazy component for that module and injects UserService, it finds a UserService provider in the lazy module's *child injector* and creates a *new* instance of the UserService. This is an entirely different UserService instance than the app-wide singleton version that Angular injected in one of the eagerly loaded components.

This scenario causes your app to create a new instance every time, instead of using the singleton.

Read More: <https://angular.io/guide/ngmodule-faq>

Pipes

A pipe takes in data as input and transforms it to a desired output. In this page, you'll use pipes to transform a component's birthday property into a human-friendly date.

src/app/hero-birthday1.component.ts

import { [Component](https://angular.io/api/core/Component) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-hero-birthday',

template: `<p>The hero's birthday is {{ birthday | date }}</p>`

})

export class HeroBirthdayComponent {

birthday = new Date(1988, 3, 15); // April 15, 1988

}

## **Built-in pipes**

Angular comes with a stock of pipes such as [DatePipe](https://angular.io/api/common/DatePipe), [UpperCasePipe](https://angular.io/api/common/UpperCasePipe), [LowerCasePipe](https://angular.io/api/common/LowerCasePipe), [CurrencyPipe](https://angular.io/api/common/CurrencyPipe), and [PercentPipe](https://angular.io/api/common/PercentPipe). They are all available for use in any template.

Angular doesn't have a FilterPipe or an OrderByPipe. Angular doesn't offer such pipes because they perform poorly and prevent aggressive minification. Both filter and orderBy require parameters that reference object properties.

## **Parameterizing a pipe**

A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon ( : ) and then the parameter value (such as currency:'EUR'). If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5)

src/app/app.component.html

<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>

## **Chaining pipes**

You can chain pipes together in potentially useful combinations.

src/app/app.component.html

The chained hero's birthday is

{{ birthday | date | [uppercase](https://angular.io/api/common/UpperCasePipe)}}

## **Custom pipes**

You can write your own custom pipes. Here's a custom pipe named ExponentialStrengthPipe that can boost a hero's powers:

src/app/exponential-strength.pipe.ts

import { [Pipe](https://angular.io/api/core/Pipe), [PipeTransform](https://angular.io/api/core/PipeTransform) } from '@angular/core';

/\*

\* Raise the value exponentially

\* Takes an exponent argument that defaults to 1.

\* Usage:

\* value | exponentialStrength:exponent

\* Example:

\* {{ 2 | exponentialStrength:10 }}

\* formats to: 1024

\*/

@[Pipe](https://angular.io/api/core/Pipe)({name: 'exponentialStrength'})

export class ExponentialStrengthPipe implements [PipeTransform](https://angular.io/api/core/PipeTransform) {

transform(value: number, exponent: string): number {

let exp = parseFloat(exponent);

return Math.pow(value, isNaN(exp) ? 1 : exp);

}

}

This pipe definition reveals the following key points:

* A pipe is a class decorated with pipe metadata.
* The pipe class implements the [PipeTransform](https://angular.io/api/core/PipeTransform) interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.
* There will be one additional argument to the transform method for each parameter passed to the pipe. Your pipe has one such parameter: the exponent.
* To tell Angular that this is a pipe, you apply the @[Pipe](https://angular.io/api/core/Pipe) decorator, which you import from the core Angular library.
* The @[Pipe](https://angular.io/api/core/Pipe) decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier. Your pipe's name is exponentialStrength.

Now you need a component to demonstrate the pipe.

src/app/power-booster.component.ts

import { [Component](https://angular.io/api/core/Component) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-power-booster',

template: `

<h2>Power Booster</h2>

<p>Super power boost: {{2 | exponentialStrength: 10}}</p>

`

})

export class PowerBoosterComponent { }

## **Pure and impure pipes**

### **Pure pipes**

Angular executes a *pure pipe* only when it detects a *pure change* to the input value. A pure change is either a change to a primitive input value (String, Number, Boolean, Symbol) or a changed object reference (Date, Array, Function, Object).

Angular ignores changes within (composite) objects. It won't call a pure pipe if you change an input month, add to an input array, or update an input object property.

This may seem restrictive but it's also fast. An object reference check is fast—much faster than a deep check for differences—so Angular can quickly determine if it can skip both the pipe execution and a view update.

### **Impure pipes**

Angular executes an *impure pipe* during every component change detection cycle. An impure pipe is called often, as often as every keystroke or mouse-move.

With that concern in mind, implement an impure pipe with great care. An expensive, long-running pipe could destroy the user experience.

Attribute Directives

An **Attribute** directive changes the appearance or behavior of a DOM element.

## **Directives overview**

There are three kinds of directives in Angular:

1. Components—directives with a template.
2. Structural directives—change the DOM layout by adding and removing DOM elements.
3. Attribute directives—change the appearance or behavior of an element, component, or another directive.

## **Build a simple attribute directive**

src/app/highlight.directive.ts

import { [Directive](https://angular.io/api/core/Directive), [ElementRef](https://angular.io/api/core/ElementRef) } from '@angular/core';

@[Directive](https://angular.io/api/core/Directive)({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(el: [ElementRef](https://angular.io/api/core/ElementRef)) {}

@[HostListener](https://angular.io/api/core/HostListener)('mouseenter') onMouseEnter() {

this.highlight('yellow');

}

@[HostListener](https://angular.io/api/core/HostListener)('mouseleave') onMouseLeave() {

this.highlight(null);

}

private highlight(color: string) {

this.el.nativeElement.style.backgroundColor = color;

}

}

Form Validation

Improve overall data quality by validating user input for accuracy and completeness.

## **Template-driven validation**

To add validation to a template-driven form, you add the same validation attributes as you would with [native HTML form validation](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation). Angular uses directives to match these attributes with validator functions in the framework.

template/hero-form-template.component.html (name)

<input id="name" name="name" class="form-control"

required [minlength](https://angular.io/api/forms/MinLengthValidator)="4" appForbiddenName="bob"

[([ngModel](https://angular.io/api/forms/NgModel))]="hero.name" #name="[ngModel](https://angular.io/api/forms/NgModel)" >

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.invalid && (name.dirty || name.touched)"

class="alert alert-danger">

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors.required">

Name is required.

</div>

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors.minlength">

Name must be at least 4 characters long.

</div>

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors.forbiddenName">

Name cannot be Bob.

</div>

</div>

Note the following:

* The <input> element carries the HTML validation attributes: required and [minlength](https://angular.io/api/forms/MinLengthValidator). It also carries a custom validator directive, forbiddenName. For more information, see [Custom validators](https://angular.io/guide/form-validation#custom-validators) section.
* #name="[ngModel](https://angular.io/api/forms/NgModel)" exports [NgModel](https://angular.io/api/forms/NgModel) into a local variable called name. [NgModel](https://angular.io/api/forms/NgModel) mirrors many of the properties of its underlying [FormControl](https://angular.io/api/forms/FormControl) instance, so you can use this in the template to check for control states such as valid and dirty. For a full list of control properties, see the [AbstractControl](https://angular.io/api/forms/AbstractControl) API reference.
* The \*[ngIf](https://angular.io/api/common/NgIf) on the <div> element reveals a set of nested message divs but only if the name is invalid and the control is either dirty or touched.
* Each nested <div> can present a custom message for one of the possible validation errors. There are messages for required, [minlength](https://angular.io/api/forms/MinLengthValidator), and forbiddenName.

## **Reactive form validation**

Angular offers two form-building technologies: reactive forms and template-driven forms. The two technologies belong to the @angular/forms library and share a common set of form control classes.

But they diverge markedly in philosophy, programming style, and technique. They even have their own modules: the [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule) and the [FormsModule](https://angular.io/api/forms/FormsModule).

### **Reactive forms**

Angular reactive forms facilitate a reactive style of programming that favors explicit management of the data flowing between a non-UI data model (typically retrieved from a server) and a UI-oriented form model that retains the states and values of the HTML controls on screen. Reactive forms offer the ease of using reactive patterns, testing, and validation.

With reactive forms, you create a tree of Angular form control objects in the component class and bind them to native form control elements in the component template, using techniques described in this guide.

### **Template-driven forms**

Template-driven forms, introduced in the [Template guide](https://angular.io/guide/forms), take a completely different approach.

You place HTML form controls (such as <input> and <select>) in the component template and bind them to data model properties in the component, using directives like [ngModel](https://angular.io/api/forms/NgModel).

You don't create Angular form control objects. Angular directives create them for you, using the information in your data bindings. You don't push and pull data values. Angular handles that for you with [ngModel](https://angular.io/api/forms/NgModel). Angular updates the mutable data model with user changes as they happen.