Table of Contents

[Setting up the local environment and workspace 2](#_Toc49921666)

[Introduction to Angular concepts 3](#_Toc49921667)

[**Modules** 3](#_Toc49921668)

[**Components** 4](#_Toc49921669)

[**Templates, directives, and data binding** 4](#_Toc49921670)

[**Services and dependency injection** 4](#_Toc49921671)

[**Routing** 4](#_Toc49921672)

[**Introduction to modules** 6](#_Toc49921673)

[**NgModule metadata** 6](#_Toc49921674)

[**NgModules and components** 7](#_Toc49921675)

[**Angular libraries** 8](#_Toc49921676)

[**Component metadata** 10](#_Toc49921677)

[**Templates and views** 11](#_Toc49921678)

[**Template syntax** 11](#_Toc49921679)

[**Data binding** 11](#_Toc49921680)

[**Create the sample project – Book Store** 13](#_Toc49921681)

[**Pipes** 17](#_Toc49921682)

[**Directives** 18](#_Toc49921683)

[**Introduction to services and dependency injection** 19](#_Toc49921684)

[**Dependency injection (DI)** 20](#_Toc49921685)

[**Providing services** 21](#_Toc49921686)

# Setting up the local environment and workspace

To install Angular on your local system, you need the following:

* **Node.js**
* **npm package manager**

Install Angular CLI

You use the Angular CLI to create projects, generate application and library code, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

To install the Angular CLI, open a terminal window and run the following command:

npm install -g @angular/cli

To create a new workspace and initial starter app:

1. Run the CLI command ng new and provide the name my-app, as shown here:

ng new my-app

1. The ng new command prompts you for information about features to include in the initial app. Accept the defaults by pressing the Enter or Return key.

The Angular CLI installs the necessary Angular npm packages and other dependencies. This can take a few minutes.

The CLI creates a new workspace and a simple Welcome app, ready to run.

#### **Running the application**

The Angular CLI includes a server, so that you can build and serve your app locally.

1. Navigate to the workspace folder, such as my-app.
2. Run the following command:

cd my-app

ng serve –open

# Introduction to Angular concepts

Angular is a platform and framework for building single-page client applications using HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The architecture of an Angular application relies on certain fundamental concepts. The basic building blocks are NgModules, which provide a compilation context for components. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

App

Services

Component

Component

Component

1.Class

2.Template

3.styles

1.Class

2.Template

3.styles

1.Class

2.Template

3.styles

## **Modules**

Angular *NgModules* differ from and complement JavaScript (ES2015) modules. An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a *root module*, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

## **Components**

Every Angular application has at least one component, the root component that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

The @[Component](https://angular.io/api/core/Component)() decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.

Decorators are functions that modify JavaScript classes. Angular defines a number of decorators that attach specific kinds of metadata to classes, so that the system knows what those classes mean and how they should work.

### **Templates, directives, and data binding**

A template combines HTML with Angular markup that can modify HTML elements before they are displayed. Template *directives* provide program logic, and *binding markup* connects your application data and the DOM. There are two types of data binding:

* *Event binding* lets your app respond to user input in the target environment by updating your application data.
* *Property binding* lets you interpolate values that are computed from your application data into the HTML.

Before a view is displayed, Angular evaluates the directives and resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports *two-way data binding*, meaning that changes in the DOM, such as user choices, are also reflected in your program data.

## **Services and dependency injection**

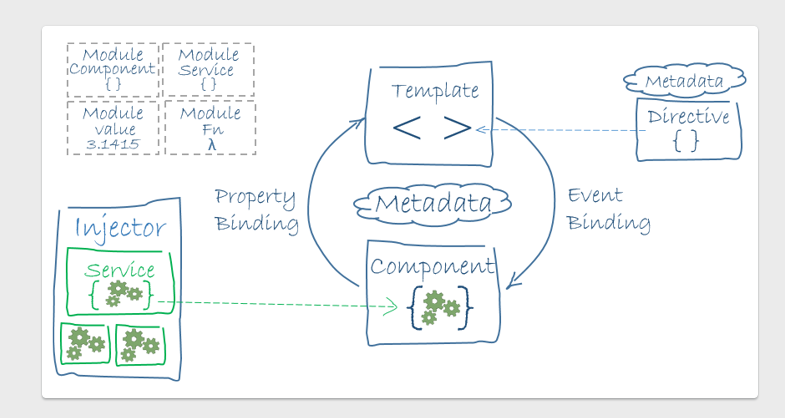
For data or logic that isn't associated with a specific view, and that you want to share across components, you create a *service* class. A service class definition is immediately preceded by the @[Injectable](https://angular.io/api/core/Injectable)() decorator. The decorator provides the metadata that allows other providers to be injected as dependencies into your class.

*Dependency injection* (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

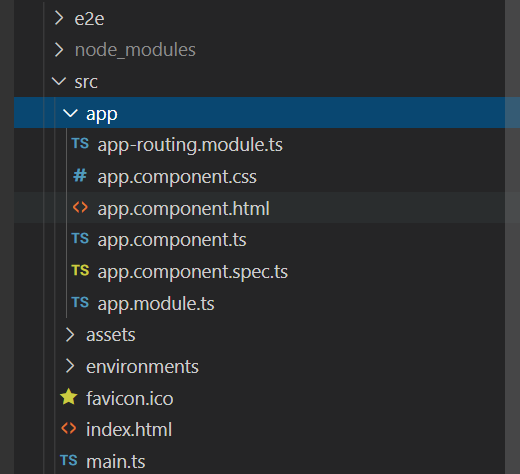
### **Routing**

The Angular [Router](https://angular.io/api/router/Router) NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app. It is modeled on the familiar browser navigation conventions:

* Enter a URL in the address bar and the browser navigates to a corresponding page.
* Click links on the page and the browser navigates to a new page.
* Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.



Directory Structure –



# **Introduction to modules**

Angular apps are modular and Angular has its own modularity system called *NgModules*. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular app has at least one NgModule class, [the *root module*](https://angular.io/guide/bootstrapping), which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by *bootstrapping* the root NgModule.

## **NgModule metadata**

An NgModule is defined by a class decorated with @[NgModule](https://angular.io/api/core/NgModule)(). The @[NgModule](https://angular.io/api/core/NgModule)() decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

* declarations: The [components](https://angular.io/guide/architecture-components), directives, and pipes that belong to this NgModule.
* exports: The subset of declarations that should be visible and usable in the component templates of other NgModules.
* imports: Other modules whose exported classes are needed by component templates declared in this NgModule.
* providers: Creators of [services](https://angular.io/guide/architecture-services) that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
* bootstrap: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

Sample root Module- src/app/app.module.ts

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

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

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

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

providers: [ Logger ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

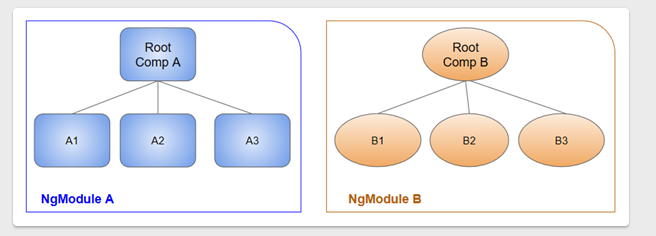
bootstrap: [ AppComponent ]

})

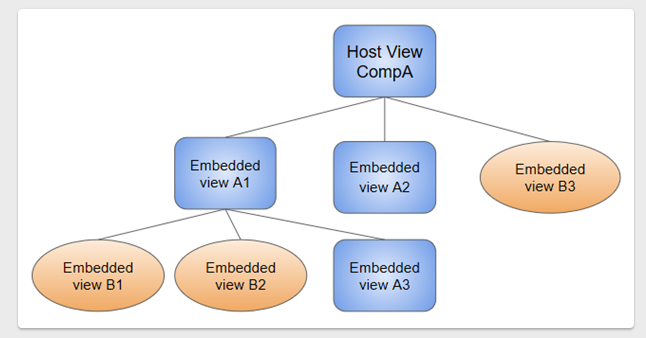
export class AppModule { }

## **NgModules and components**

NgModules provide a compilation context for their components. A root NgModule always has a root component that is created during bootstrap, but any NgModule can include any number of additional components, which can be loaded through the router or created through the template. The components that belong to an NgModule share a compilation context.



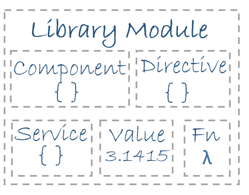
A component and its template together define a view. A component can contain a view hierarchy, which allows you to define arbitrarily complex areas of the screen that can be created, modified, and destroyed as a unit. A view hierarchy can mix views defined in components that belong to different NgModules.



When you create a component, it's associated directly with a single view, called the host view. The host view can be the root of a view hierarchy, which can contain embedded views, which are in turn the host views of other components. Those components can be in the same NgModule, or can be imported from other NgModules. Views in the tree can be nested to any depth.

## **Angular libraries**

Angular loads as a collection of JavaScript modules. You can think of them as library modules. Each Angular library name begins with the @angular prefix. Install them with the node package manager npm and import parts of them with JavaScript import statements



For example, import Angular's [Component](https://angular.io/api/core/Component) decorator from the @angular/core library like this.

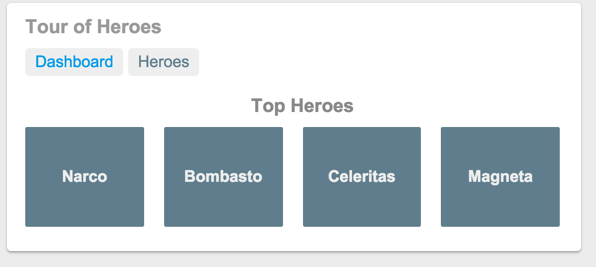
import { [Component](https://angular.io/api/core/Component) } from '@angular/core';

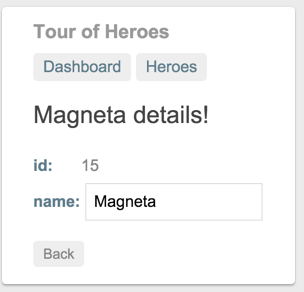
Introduction to components and templates

A *component* controls a patch of screen called a [*view*](https://angular.io/guide/glossary#view).

For example, individual components define and control each of the following views from the Hero App - ng generate component hero

* The app root with the navigation links.
* The list of heroes.
* The hero editor.





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](https://angular.io/guide/lifecycle-hooks), like ngOnInit().

## **Component metadata**

The @[Component](https://angular.io/api/core/Component) decorator identifies the class immediately below it as a component class, and specifies its metadata. In the example code below, you can see that HeroListComponent is just a class, with no special Angular notation or syntax at all. It's not a component until you mark it as one with the @[Component](https://angular.io/api/core/Component) decorator.

Here's an example of basic 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) { /\* . . . \*/ }

This example shows some of the most useful @[Component](https://angular.io/api/core/Component) configuration options:

* selector: A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains <app-hero-list></app-hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.
* templateUrl: The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the template property. This template defines the component's host view.
* providers: An array of [providers](https://angular.io/guide/glossary#provider) for services that the component requires. In the example, this tells Angular how to provide the HeroService instance that the component's constructor uses to get the list of heroes to display.

## **Templates and views**

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.

## **Template syntax**

<h2>Hero List</h2>

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

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

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

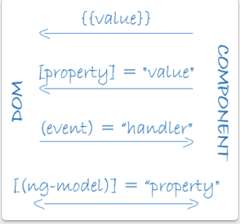
This template uses typical HTML elements like <h2> and <p>, and also includes Angular template-syntax elements, \*[ngFor](https://angular.io/api/common/NgForOf), {{hero.name}}, (click), [hero], and <app-hero-detail>. The template-syntax elements tell Angular how to render the HTML to the screen, using program logic and data.

### **Data binding**

Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push and pull logic by hand is tedious, error-prone, and a nightmare to read, as any experienced front-end JavaScript programmer can attest.

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

The following diagram shows the four forms of data binding markup. Each form has a direction: to the DOM, from the DOM, or both.



This example from the HeroListComponent template uses three of these forms.

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

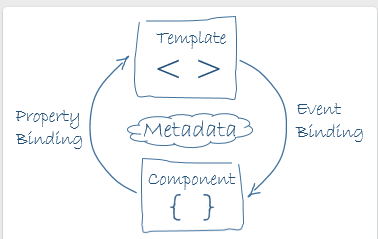
<app-hero-detail [hero]="selectedHero"></app-hero-detail> <li (click)="selectHero(hero)"></li>

* 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/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#binding-to-user-input-events) calls the component's selectHero method when the user clicks a hero's name.

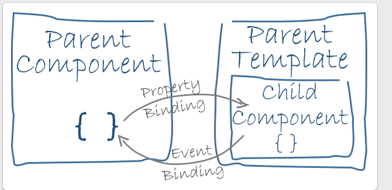
Two-way data binding (used mainly in [template-driven forms](https://angular.io/guide/forms)) combines property and event binding in a single notation. Here's an example from the HeroDetailComponent template that uses two-way data binding with the [ngModel](https://angular.io/api/forms/NgModel) directive.

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

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



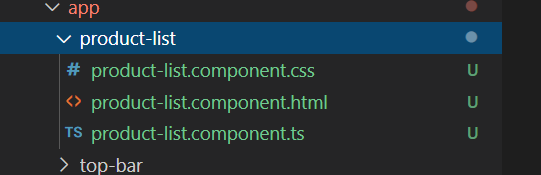
Data binding plays an important role in communication between a template and its component, and is also important for communication between parent and child components.



## **Create the sample project – Book Store**

Requirements –

* Component
  1. Template
  2. Component
  3. Style



Once created need to add to Main App Module

Template -> product-list.component.html

<div class="product-list">

    <h2>Available Books</h2>

</div>

Styles-> product-list.component.css

.product-list{

    background-color: #c2c2c2;

}

Component.ts

import { Component, OnInit } from '@angular/core';

import { products } from '../products';

@Component({

    selector: 'app-book-list',

    styleUrls: ['./product-list.component.css'],

    templateUrl: './product-list.component.html'

})

export class ProductListComponent implements OnInit {

    products = products;

    constructor(){}

    ngOnInit(): void{

        // TODO

    }

}

Now we will have to add this newly component to main module -> app.module.ts

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

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

import { AppRoutingModule } from './app-routing.module';

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

import {ProductListComponent} from './product-list/product-list.component';

@NgModule({

  declarations: [

    AppComponent,

    ProductListComponent // added Component

  ],

  imports: [

    BrowserModule,

    AppRoutingModule

  ],

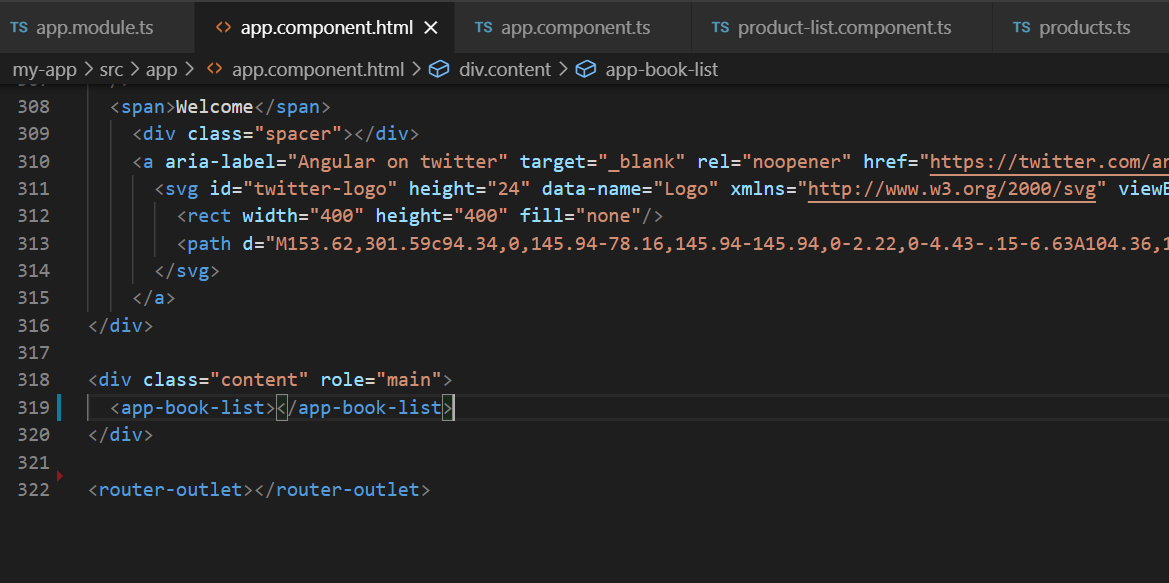
  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

Then add newly created template to app templates –

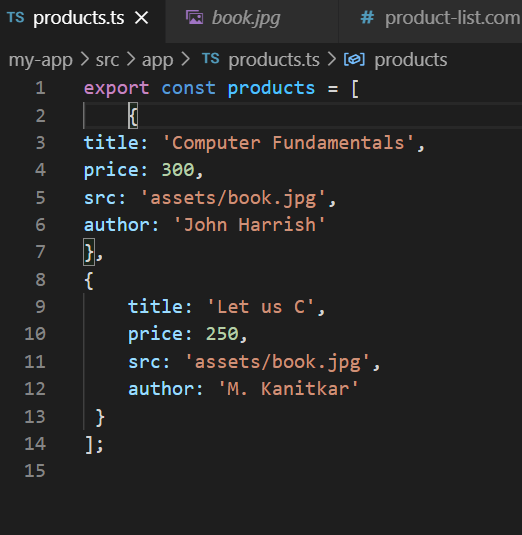


Add Bootstrap for styling add globally

* npm install –save bootstrap -- it will add bootstrap into node\_modules
* add it to global css file as below

@import "~bootstrap/dist/css/bootstrap.css";

Let us add some data to our Book Store



Update bookstore template to use data

<div class="container product-list">

    <h2>Available Books</h2>

    <div class="row">

        <div class="col-md-6" \*ngFor="let product of products">

            <div class="card" style="width: 18rem;">

                <img src="{{product.src}}" class="card-img-top" alt="...">

                <div class="card-body">

                  <h5 class="card-title">{{product.title}}</h5>

                  <p class="card-text">{{product.author}}</p>

                  <p class="card-text">Price: {{product.price | currency}}</p>

                  <a href="#" class="btn btn-primary" [title]="product.title">Go somewhere</a>

                </div>

              </div>

        </div>

    </div>

</div>

To make each product name a link to product details, add the <a> element and set its title to be the product's name by using the property binding [ ] syntax, as above

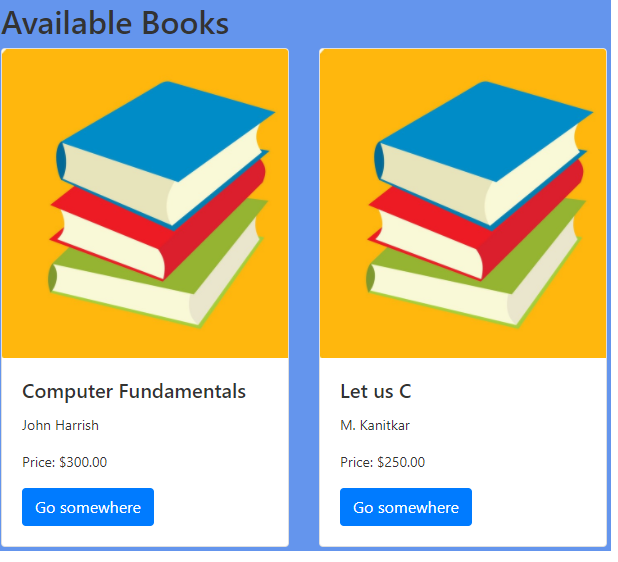
To check, hold the pointer over a product button to see the bound name property value, which is the product name plus the word "details". Interpolation {{ }} lets you render the property value as text; property binding [ ] lets you use the property value in a template expression.

Add the product descriptions. On the <p> element, use an \*[ngIf](https://angular.io/api/common/NgIf) directive so that Angular only creates the <p> element if the current product has a description.

<p \*[ngIf](https://angular.io/api/common/NgIf)="product.description"> Description: {{ product.description }} </p>

Add a button so users can share a product with friends. Bind the button's click event to the share() method (in product-list.component.ts). Event binding uses a set of parentheses, ( ), around the event, as in the following <button> element:

<button (click)="share()"> Share </button>



The app now has a product list and sharing feature. In the process, you've learned to use five common features of Angular's template syntax:

* \*[ngFor](https://angular.io/api/common/NgForOf)
* \*[ngIf](https://angular.io/api/common/NgIf)
* Interpolation {{ }}
* Property binding [ ]
* Event binding ( )

### **Pipes**

Angular pipes let you declare display-value transformations in your template HTML. A class with the @[Pipe](https://angular.io/api/core/Pipe) decorator defines a function that transforms input values to output values for display in a view.

Angular defines various pipes, such as the [date](https://angular.io/api/common/DatePipe) pipe and [currency](https://angular.io/api/common/CurrencyPipe) pipe; for a complete list, see the [Pipes API list](https://angular.io/api?type=pipe). You can also define new pipes.

To specify a value transformation in an HTML template, use the [pipe operator (|)](https://angular.io/guide/template-expression-operators#pipe).

{{interpolated\_value | pipe\_name}}

You can chain pipes, sending the output of one pipe function to be transformed by another pipe function. A pipe can also take arguments that control how it performs its transformation. For example, you can pass the desired format to the [date](https://angular.io/api/common/DatePipe) pipe.

<!-- Default format: output 'Jun 15, 2020'-->

<p>Today is {{today | [date](https://angular.io/api/common/DatePipe)}}</p>

<!-- fullDate format: output 'Monday, June 15, 2020'-->

<p>The date is {{today | [date](https://angular.io/api/common/DatePipe):'fullDate'}}</p>

<!-- shortTime format: output '9:43 AM'-->

<p>The time is {{today | [date](https://angular.io/api/common/DatePipe):'shortTime'}}</p>

### **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 technically a directive. However, components are so distinctive and central to Angular applications that Angular defines the @[Component](https://angular.io/api/core/Component)() decorator, which extends the @[Directive](https://angular.io/api/core/Directive)() decorator with template-oriented features.

In addition to components, there are two other kinds of directives: structural and attribute. Angular defines a number of directives of both kinds, and you can define your own using the @[Directive](https://angular.io/api/core/Directive)() decorator.

Just as for components, the metadata for a directive associates the decorated class with a selector element that you use to insert it into HTML. In templates, directives typically appear within an element tag as attributes, either by name or as the target of an assignment or a binding.

#### **Structural directives**

Structural directives alter layout by adding, removing, and replacing elements in the DOM. The example template uses two built-in structural directives to add application logic to how the view is rendered.

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

#### **Attribute directives**

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 <input>) by setting its display value property and responding to change events.

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

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

# **Introduction to services and dependency injection**

Service is a broad category encompassing any value, function, or feature that an app needs. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

Ideally, a component's job is to enable the user experience and nothing more. A component should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).

A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an injectable service class, you make those tasks available to any component. You can also make your app more adaptable by injecting different providers of the same kind of service, as appropriate in different circumstances.

Angular doesn't enforce these principles. Angular does help you follow these principles by making it easy to factor your application logic into services and make those services available to components through dependency injection.

Example –

export class Logger {

log(msg: any) {

console.log(msg);

}

error(msg: any) {

console.error(msg);

}

warn(msg: any) {

console.warn(msg);

}

}

Services can depend on other services. For example, here's a HeroService that depends on the Logger service, and also uses BackendService to get heroes. That service in turn might depend on the [HttpClient](https://angular.io/api/common/http/HttpClient) service to fetch heroes asynchronously from a server.

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;

}

}

## **Dependency injection (DI)**

DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can inject a service into a component, giving the component access to that service class.

To define a class as a service in Angular, use the @[Injectable](https://angular.io/api/core/Injectable)() decorator to provide the metadata that allows Angular to inject it into a component as a dependency. Similarly, use the @[Injectable](https://angular.io/api/core/Injectable)() decorator to indicate that a component or other class (such as another service, a pipe, or an NgModule) has a dependency.

* The injector is the main mechanism. Angular creates an application-wide injector for you during the bootstrap process, and additional injectors as needed. You don't have to create injectors.
* An injector creates dependencies, and maintains a container of dependency instances that it reuses if possible.
* A provider is an object that tells an injector how to obtain or create a dependency.

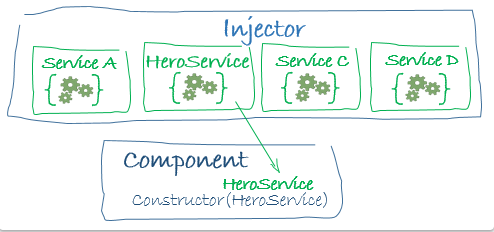
constructor(private service: HeroService) { }

When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the constructor parameter types. For example, the constructor of HeroListComponent needs HeroService.

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector makes one using the registered provider, and adds it to the injector 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.

The process of HeroService injection looks something like this.



### **Providing services**

You must register at least one provider of any service you are going to use. The provider can be part of the service's own metadata, making that service available everywhere, or you can register providers with specific modules or components. You register providers in the metadata of the service (in the @[Injectable](https://angular.io/api/core/Injectable)() decorator), or in the @[NgModule](https://angular.io/api/core/NgModule)() or @[Component](https://angular.io/api/core/Component)() metadata

When you provide the service at the root level, Angular creates a single, shared instance of HeroService and injects it into any class that asks for it. Registering the provider in the @[Injectable](https://angular.io/api/core/Injectable)() metadata also allows Angular to optimize an app by removing the service from the compiled app if it isn't used.

@[Injectable](https://angular.io/api/core/Injectable)({ providedIn: 'root', })

When you register a provider with a [specific NgModule](https://angular.io/guide/architecture-modules), the same instance of a service is available to all components in that NgModule. To register at this level, use the providers property of the @[NgModule](https://angular.io/api/core/NgModule)() decorator,

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

providers: [ BackendService, Logger ],

...

})

When you register a provider at the component level, you get a new instance of the service with each new instance of that component. At the component level, register a service provider in the providers property of the @[Component](https://angular.io/api/core/Component)() metadata.

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

selector: 'app-hero-list',

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

providers: [ HeroService ]

})

# **Introduction to forms in Angular**

Handling user input with forms is the cornerstone of many common applications. Applications use forms to enable users to log in, to update a profile, to enter sensitive information, and to perform many other data-entry tasks.

Angular provides two different approaches to handling user input through forms: reactive and template-driven. Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.

## **Choosing an approach**

Reactive forms and template-driven forms process and manage form data differently. Each approach offers different advantages.

* **Reactive forms** provide direct, explicit access to the underlying forms object model. Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.
* **Template-driven forms** rely on directives in the template to create and manipulate the underlying object model. They are useful for adding a simple form to an app, such as an email list signup form. They're easy to add to an app, but they don't scale as well as reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, template-driven forms could be a good fit.

### **Common form foundation classes**

Both reactive and template-driven forms are built on the following base classes.

* [FormControl](https://angular.io/api/forms/FormControl) tracks the value and validation status of an individual form control.
* [FormGroup](https://angular.io/api/forms/FormGroup) tracks the same values and status for a collection of form controls.
* [FormArray](https://angular.io/api/forms/FormArray) tracks the same values and status for an array of form controls.
* [ControlValueAccessor](https://angular.io/api/forms/ControlValueAccessor) creates a bridge between Angular [FormControl](https://angular.io/api/forms/FormControl) instances and native DOM elements.

### **Setup in reactive forms**

With reactive forms, you define the form model directly in the component class. The [formControl] directive links the explicitly created [FormControl](https://angular.io/api/forms/FormControl) instance to a specific form element in the view, using an internal value accessor.

The following component implements an input field for a single control, using reactive forms. In this example, the form model is the [FormControl](https://angular.io/api/forms/FormControl) instance.

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

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

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

selector: 'app-reactive-favorite-color',

template: ` Favorite Color: <input type="text" [formControl]="favoriteColorControl"> `

})

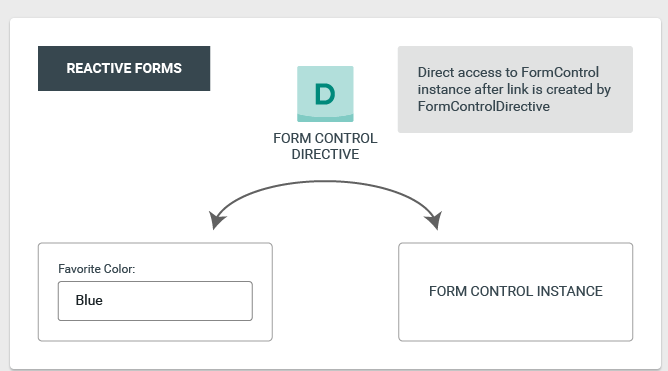
export class FavoriteColorComponent {

favoriteColorControl = new [FormControl](https://angular.io/api/forms/FormControl)('');

}

Below Figure shows how, in reactive forms, the form model is the source of truth; it provides the value and status of the form element at any given point in time, through the [formControl] directive on the input element.

*Direct access to forms model in a reactive form.*



### **Setup in template-driven forms**

In template-driven forms, the form model is implicit, rather than explicit. The directive [NgModel](https://angular.io/api/forms/NgModel) creates and manages a [FormControl](https://angular.io/api/forms/FormControl) instance for a given form element.

The following component implements the same input field for a single control, using template-driven forms.

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

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

selector: 'app-template-favorite-color',

template: ` Favorite Color: <input type="text" [([ngModel](https://angular.io/api/forms/NgModel))]="favoriteColor"> `

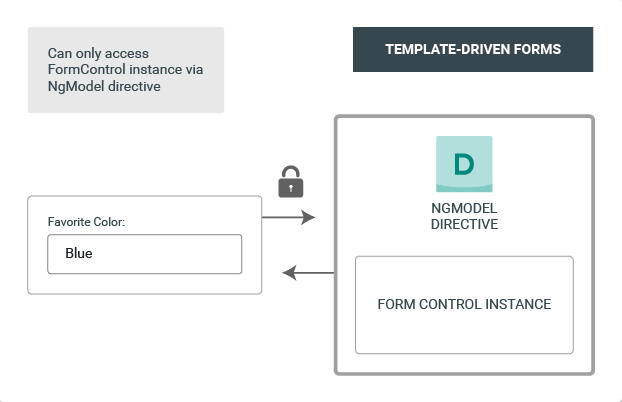
})

export class FavoriteColorComponent {

favoriteColor = '';

}

In a template-driven form the source of truth is the template. You do not have direct programmatic access to the [FormControl](https://angular.io/api/forms/FormControl) instance, as shown in below Figure



## **Data flow in forms**

When an application contains a form, Angular must keep the view in sync with the component model and the component model in sync with the view. As users change values and make selections through the view, the new values must be reflected in the data model. Similarly, when the program logic changes values in the data model, those values must be reflected in the view.

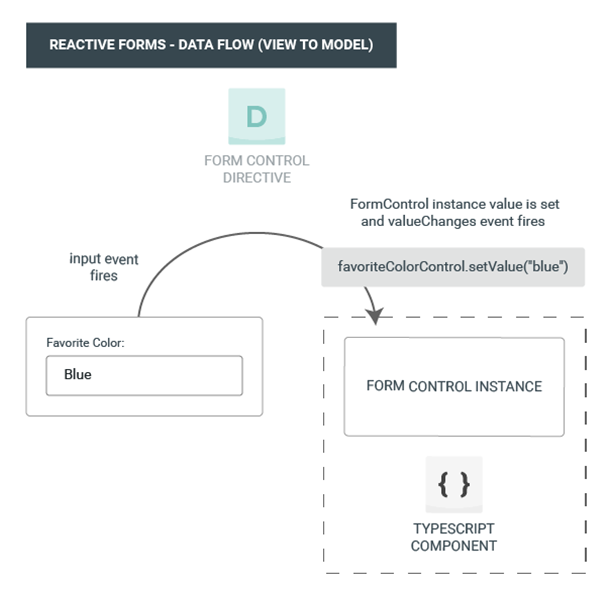
Reactive and template-driven forms differ in how they handle data flowing from the user or from programmatic changes. The following diagrams illustrate both kinds of data flow for each type of form, using the favorite-color input field defined above.

### **Data flow in reactive forms**

In reactive forms each form element in the view is directly linked to the form model (a [FormControl](https://angular.io/api/forms/FormControl) instance). Updates from the view to the model and from the model to the view are synchronous and do not depend on how the UI is rendered.

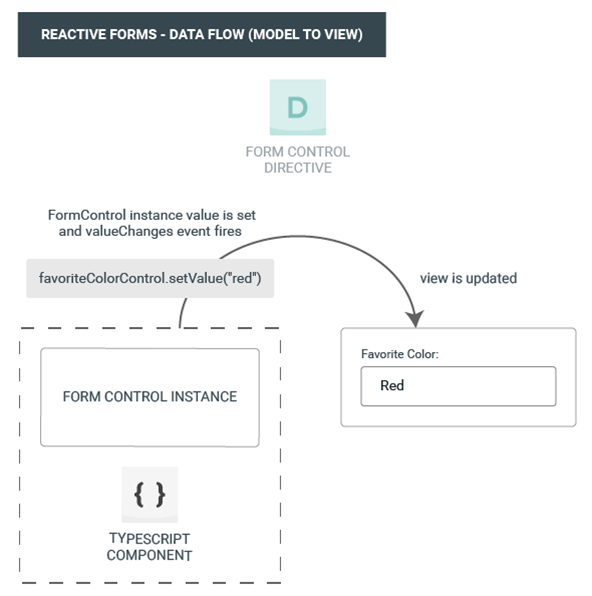
The view-to-model diagram shows how data flows when an input field's value is changed from the view through the following steps.

1. The user types a value into the input element, in this case the favorite color Blue.
2. The form input element emits an "input" event with the latest value.
3. The control value accessor listening for events on the form input element immediately relays the new value to the [FormControl](https://angular.io/api/forms/FormControl) instance.
4. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the new value through the valueChanges observable.
5. Any subscribers to the valueChanges observable receive the new value.



The model-to-view diagram shows how a programmatic change to the model is propagated to the view through the following steps.

1. The user calls the favoriteColorControl.setValue() method, which updates the [FormControl](https://angular.io/api/forms/FormControl) value.
2. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the new value through the valueChanges observable.
3. Any subscribers to the valueChanges observable receive the new value.
4. The control value accessor on the form input element updates the element with the new value.



### **Data flow in template-driven forms**

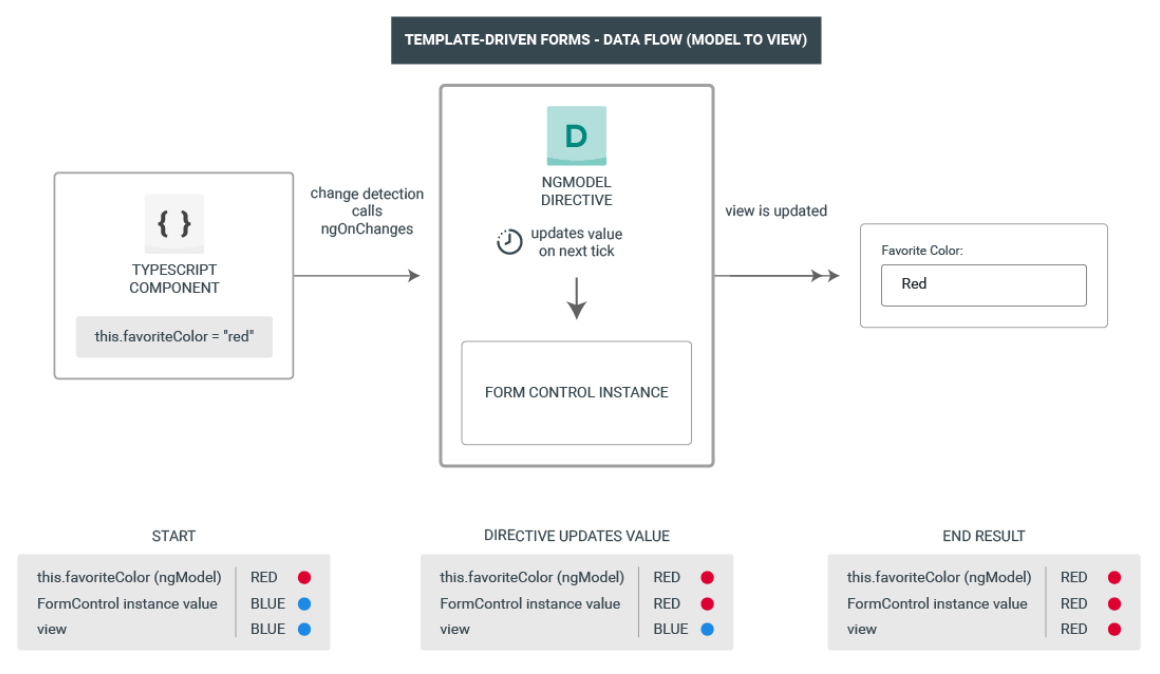
In template-driven forms, each form element is linked to a directive that manages the form model internally.

The view-to-model diagram shows how data flows when an input field's value is changed from the view through the following steps.

1. The user types Blue into the input element.
2. The input element emits an "input" event with the value Blue.
3. The control value accessor attached to the input triggers the setValue() method on the [FormControl](https://angular.io/api/forms/FormControl) instance.
4. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the new value through the valueChanges observable.
5. Any subscribers to the valueChanges observable receive the new value.
6. The control value accessor also calls the [NgModel.viewToModelUpdate()](https://angular.io/api/forms/NgModel" \l "viewToModelUpdate) method which emits an ngModelChange event.
7. Because the component template uses two-way data binding for the favoriteColor property, the favoriteColor property in the component is updated to the value emitted by the ngModelChange event (Blue).

The model-to-view diagram shows how data flows from model to view when the favoriteColor changes from Blue to Red, through the following steps

1. The favoriteColor value is updated in the component.
2. Change detection begins.
3. During change detection, the ngOnChanges lifecycle hook is called on the [NgModel](https://angular.io/api/forms/NgModel) directive instance because the value of one of its inputs has changed.
4. The ngOnChanges() method queues an async task to set the value for the internal [FormControl](https://angular.io/api/forms/FormControl) instance.
5. Change detection completes.
6. On the next tick, the task to set the [FormControl](https://angular.io/api/forms/FormControl) instance value is executed.
7. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the latest value through the valueChanges observable.
8. Any subscribers to the valueChanges observable receive the new value.
9. The control value accessor updates the form input element in the view with the latest favoriteColor value.



### **Mutability of the data model**

The change-tracking method plays a role in the efficiency of your application.

* **Reactive forms** keep the data model pure by providing it as an immutable data structure. Each time a change is triggered on the data model, the [FormControl](https://angular.io/api/forms/FormControl) instance returns a new data model rather than updating the existing data model. This gives you the ability to track unique changes to the data model through the control's observable. Change detection is more efficient because it only needs to update on unique changes. Because data updates follow reactive patterns, you can integrate with observable operators to transform data.
* **Template-driven** forms rely on mutability with two-way data binding to update the data model in the component as changes are made in the template. Because there are no unique changes to track on the data model when using two-way data binding, change detection is less efficient at determining when updates are required.

The difference is demonstrated in the previous examples that use the favorite-color input element.

* With reactive forms, the [FormControl](https://angular.io/api/forms/FormControl)**instance** always returns a new value when the control's value is updated.
* With template-driven forms, the **favorite color property** is always modified to its new value.

## **Form validation**

Validation is an integral part of managing any set of forms. Whether you're checking for required fields or querying an external API for an existing username, Angular provides a set of built-in validators as well as the ability to create custom validators.

* **Reactive forms** define custom validators as **functions** that receive a control to validate.
* **Template-driven forms** are tied to template **directives**, and must provide custom validator directives that wrap validation functions.

# **Reactive forms**

Reactive forms provide a model-driven approach to handling form inputs whose values change over time. This guide shows you how to create and update a basic form control, progress to using multiple controls in a group, validate form values, and create dynamic forms where you can add or remove controls at run time.

## **Overview of reactive forms**

Reactive forms use an explicit and immutable approach to managing the state of a form at a given point in time. Each change to the form state returns a new state, which maintains the integrity of the model between changes. Reactive forms are built around [observable](https://angular.io/guide/glossary#observable) streams, where form inputs and values are provided as streams of input values, which can be accessed synchronously.

Reactive forms also provide a straightforward path to testing because you are assured that your data is consistent and predictable when requested. Any consumers of the streams have access to manipulate that data safely.

Reactive forms differ from [template-driven forms](https://angular.io/guide/forms) in distinct ways. Reactive forms provide more predictability with synchronous access to the data model, immutability with observable operators, and change tracking through observable streams.

Template-driven forms allow direct access to modify data in your template, but are less explicit than reactive forms because they rely on directives embedded in the template, along with mutable data to track changes asynchronously. See the [Forms Overview](https://angular.io/guide/forms-overview) for detailed comparisons between the two paradigms.

## **Adding a basic form control**

There are three steps to using form controls.

1. Register the reactive forms module in your app. This module declares the reactive-form directives that you need to use reactive forms.
2. Generate a new [FormControl](https://angular.io/api/forms/FormControl) instance and save it in the component.
3. Register the [FormControl](https://angular.io/api/forms/FormControl) in the template.

**Register the reactive forms module**

To use reactive form controls, import [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule) from the @angular/forms package and add it to your NgModule's imports array.

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

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

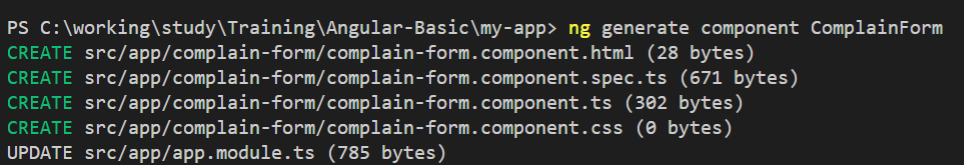
imports: [ // other imports ... [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule) ],

})

export class AppModule { }

**Generate a new**[FormControl](https://angular.io/api/forms/FormControl)

ng generate component ComplainForm



To register a single form control, import the [FormControl](https://angular.io/api/forms/FormControl) class and create a new instance of [FormControl](https://angular.io/api/forms/FormControl) to save as a class property.

Use the constructor of [FormControl](https://angular.io/api/forms/FormControl) to set its initial value, which in this case is an empty string. By creating these controls in your component class, you get immediate access to listen for, update, and validate the state of the form input.

import { Component, OnInit } from '@angular/core';

import { FormControl } from '@angular/forms';

@Component({

  selector: 'app-complain-form',

  templateUrl: './complain-form.component.html',

  styleUrls: ['./complain-form.component.css']

})

export class ComplainFormComponent implements OnInit {

  constructor() { }

  name = new FormControl('');

  ngOnInit(): void {

  }

}

**Register the control in the template**

<label> Name: <input type="text" [formControl]="name"> </label>

### **Displaying a form control value**

You can display the value in the following ways.

* Through the valueChanges observable where you can listen for changes in the form's value in the template using [AsyncPipe](https://angular.io/api/common/AsyncPipe) or in the component class using the subscribe() method.
* With the value property, which gives you a snapshot of the current value.

The following example shows you how to display the current value using interpolation in the template.

Forms typically contain several related controls. Reactive forms provide two ways of grouping multiple related controls into a single input form.

* A form group defines a form with a fixed set of controls that you can manage together. Form group basics are discussed in this section. You can also [nest form groups](https://angular.io/guide/reactive-forms#nested-groups) to create more complex forms.
* A form array defines a dynamic form, where you can add and remove controls at run time. You can also nest form arrays to create more complex forms. For more about this option, see [Creating dynamic forms](https://angular.io/guide/reactive-forms#dynamic-forms) below.

Just as a form control instance gives you control over a single input field, a form group instance tracks the form state of a group of form control instances (for example, a form). Each control in a form group instance is tracked by name when creating the form group. The following example shows how to manage multiple form control instances in a single group.

Generate a ProfileEditor component and import the [FormGroup](https://angular.io/api/forms/FormGroup) and [FormControl](https://angular.io/api/forms/FormControl) classes from the @angular/forms package.

ng generate component ProfileEditor

add formGroup and formControl to profile-editor.component.ts

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

To add a form group to this component, take the following steps.

1. Create a [FormGroup](https://angular.io/api/forms/FormGroup) instance.
2. Associate the [FormGroup](https://angular.io/api/forms/FormGroup) model and view.
3. Save the form data.

**Create a FormGroup instance**

import { Component, OnInit } from '@angular/core';

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

@Component({

  selector: 'app-profile-editor',

  templateUrl: './profile-editor.component.html',

  styleUrls: ['./profile-editor.component.css']

})

export class ProfileEditorComponent implements OnInit {

  constructor() { }

  profileForm = new FormGroup({

    name: new FormControl(''),

    issue: new FormControl('')

  });

  ngOnInit(): void {

  }

}

**Associate the FormGroup model and view**

<p>profile-editor</p>

<form [formGroup]="profileForm">

    <label>

       Name:

      <input type="text" formControlName="name">

    </label>

    <label>

      Issue:

      <input type="text" formControlName="issue">

    </label>

  </form>

**Save form data**

The ProfileEditor component accepts input from the user, but in a real scenario you want to capture the form value and make available for further processing outside the component. The [FormGroup](https://angular.io/api/forms/FormGroup) directive listens for the submit event emitted by the form element and emits an ngSubmit event that you can bind to a callback function.

Add an ngSubmit event listener to the form tag with the onSubmit() callback method.

<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">

Use a button element to add a button to the bottom of the form to trigger the form submission.

<button type="submit" [disabled]="!profileForm.valid">Submit</button>

Display the component

To display the ProfileEditor component that contains the form, add it to a component template.

<app-profile-editor></app-profile-editor>



**Next is to save Form data on Submit of form and add it to table**

**Create data for Table and whenever forms get submitted add the form data to the table data**

complaints = [];

**then update submit method as below**

onSubmit() {

    console.warn(this.profileForm.value);

    this.complaints.push(this.profileForm.value);

this.profileForm.setValue({

      name: '', issue: ''

    });

  }

**Add table to the template which will read the complaints data.**

<table class="table">

      <thead>

          <tr>

              <th>Name</th>

              <th>Issues</th>

          </tr>

      </thead>

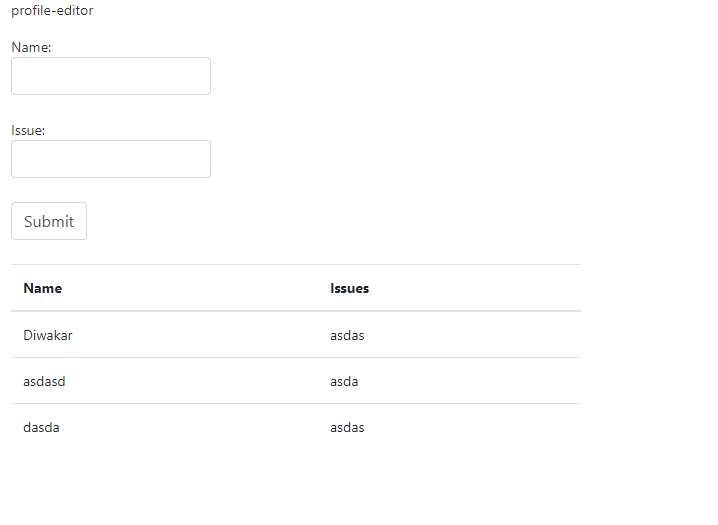
      <tbody>

          <tr \*ngFor="let item of complaints"><td>{{item.name}}</td><td>{{item.issue}}</td></tr>

      </tbody>

  </table>

**Once we submit the form and table get updated we need to reset the form**



## **Validating form input**

Form validation is used to ensure that user input is complete and correct. This section covers adding a single validator to a form control and displaying the overall form status. Form validation is covered more extensively in the [Form Validation](https://angular.io/guide/form-validation) guide.

Use the following steps to add form validation.

1. Import a validator function in your form component.
2. Add the validator to the field in the form.
3. Add logic to handle the validation status.

The most common validation is making a field required. The following example shows how to add a required validation to the name control and display the result of validation.

Import a validator function

Import the [Validators](https://angular.io/api/forms/Validators) class from the @angular/forms package.

import { Component, OnInit } from '@angular/core';

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

import { Validators } from '@angular/forms';

@Component({

  selector: 'app-profile-editor',

  templateUrl: './profile-editor.component.html',

  styleUrls: ['./profile-editor.component.css']

})

export class ProfileEditorComponent implements OnInit {

  constructor() {  }

  submitted = false;

  profileForm = new FormGroup({

    name: new FormControl('', Validators.required),

    issue: new FormControl('', Validators.required)

  });

  complaints = [];

  // tslint:disable-next-line:typedef

  onSubmit() {

    this.submitted = true;

    if (this.profileForm.invalid) {

      return;

    }

    console.warn(this.profileForm.value);

    this.complaints.push(this.profileForm.value);

    this.profileForm.setValue({

      name: '', issue: ''

    });

    this.submitted = false;

  }

  // tslint:disable-next-line:typedef

  get form() {

     return this.profileForm.controls;

  }

  ngOnInit(): void {

  }

}

<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">

    <div class="form-group">

    <label>

       Name:

      <input type="text" formControlName="name" class="form-control" required [ngClass]="{ 'is-invalid': submitted && form.name.errors }" >

      <div \*ngIf="submitted && form.name.errors && form.name.errors.required" class="invalid-feedback">

        name is required.

      </div>

    </label>

    </div>

    <div class="form-group">

    <label>

      Issue:

      <input type="text" formControlName="issue" class="form-control" required [ngClass]="{ 'is-invalid': submitted && form.issue.errors }" >

      <div \*ngIf="submitted && form.issue.errors && form.issue.errors.required" class="invalid-feedback">

        Issue is required.

      </div>

    </label>

    </div>

    <div class="form-group">

    <label>

        <button type="submit"  class="form-control">Submit</button>

    </label>

    </div>

  </form>

In-app navigation: routing to views

In a single-page app, you change what the user sees by showing or hiding portions of the display that correspond to particular components, rather than going out to the server to get a new page. As users perform application tasks, they need to move between the different [views](https://angular.io/guide/glossary#view) that you have defined.

To handle the navigation from one [view](https://angular.io/guide/glossary#view) to the next, you use the Angular [Router](https://angular.io/api/router/Router). The [Router](https://angular.io/api/router/Router) enables navigation by interpreting a browser URL as an instruction to change the view.

#### **<base href>**

This guide works with a CLI-generated Angular app. If you are working manually, make sure that you have <base href="/"> in the <head> of your index.html file. This assumes that the app folder is the application root, and uses "/".

Lets use existing component for the routing

### **Importing all components which we want use in routing**

import {ProductListComponent} from './product-list/product-list.component';

import { ProfileEditorComponent } from './profile-editor/profile-editor.component';

## **Defining a basic route**

There are three fundamental building blocks to creating a route.

Import the AppRoutingModule into AppModule and add it to the imports array.

The Angular CLI performs this step for you. However, if you are creating an app manually or working with an existing, non-CLI app, verify that the imports and configuration are correct. The following is the default AppModule using the CLI with the --routing flag.

ng new routing-app –-routing

Since we are adding routing to existing app we need to create routing module

First thing we need to import routing into our main module as below

App.module.ts

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

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

import { AppRoutingModule } from './app-routing.module';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import {ProductListComponent} from './product-list/product-list.component';

import { ProfileEditorComponent } from './profile-editor/profile-editor.component';

@NgModule({

  declarations: [

    AppComponent,

    ProductListComponent,

    ProfileEditorComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule,

    ReactiveFormsModule

  ],

  providers: [],

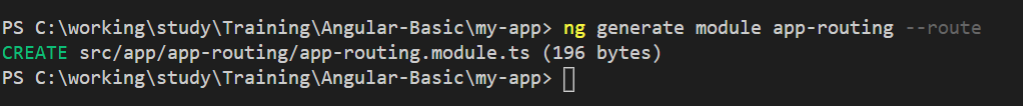
  bootstrap: [AppComponent]

})

export class AppModule { }

next step is to create routing module as below

ng generate **module app-routing –route**

while will create file - app-routing.module.ts

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

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

const routes: Routes = [];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

add it will add reference into main appModule as

import { AppRoutingModule } from './app-routing.module';

and it will update import in appModule as below

imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule,

    ReactiveFormsModule

  ],

Final AppModule will look like

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

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

import { AppRoutingModule } from './app-routing.module';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import {ProductListComponent} from './product-list/product-list.component';

import { ProfileEditorComponent } from './profile-editor/profile-editor.component';

@NgModule({

  declarations: [

    AppComponent,

    ProductListComponent,

    ProfileEditorComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule,

    ReactiveFormsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

Next step: -

* 1. Import [RouterModule](https://angular.io/api/router/RouterModule) and [Routes](https://angular.io/api/router/Routes) into your routing module.

The Angular CLI performs this step automatically. The CLI also sets up a [Routes](https://angular.io/api/router/Routes) array for your routes and configures the imports and exports arrays for @[NgModule](https://angular.io/api/core/NgModule)().

* 1. Define your routes in your [Routes](https://angular.io/api/router/Routes) array.

Each route in this array is a JavaScript object that contains two properties. The first property, path, defines the URL path for the route. The second property, component, defines the component Angular should use for the corresponding path.

import {ProductListComponent} from './product-list/product-list.component';

import { ProfileEditorComponent } from './profile-editor/profile-editor.component';

const routes: Routes = [

  {path: 'product-list', component: ProductListComponent},

  {path: 'complaint-list', component: ProfileEditorComponent},

];

* 1. Add your routes to your application.

Now that you have defined your routes, you can add them to your application. First, add links to the two components. Assign the anchor tag that you want to add the route to the [routerLink](https://angular.io/api/router/RouterLink) attribute. Set the value of the attribute to the component to show when a user clicks on each link. Next, update your component template to include <[router-outlet](https://angular.io/api/router/RouterOutlet)>. This element informs Angular to update the application view with the component for the selected route.

<nav>

    <ul>

      <li><a routerLink="/product-list" routerLinkActive="active">Book List</a></li>

      <li><a routerLink="/complaint-list" routerLinkActive="active">Issues</a></li>

    </ul>

  </nav>

<!-- The routed views render in the <[router-outlet](https://angular.io/api/router/RouterOutlet)>-->

<[router-outlet](https://angular.io/api/router/RouterOutlet)></[router-outlet](https://angular.io/api/router/RouterOutlet)>

## **Setting up wildcard routes**

A well-functioning application should gracefully handle when users attempt to navigate to a part of your application that does not exist. To add this functionality to your application, you set up a wildcard route. The Angular router selects this route any time the requested URL doesn't match any router paths.

To set up a wildcard route, add the following code to your routes definition.

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

Generate a pageNotfound and include that here

ng generate component pageNotFound

## **Setting up redirects**

To set up a redirect, configure a route with the path you want to redirect from, the component you want to redirect to, and a pathMatch value that tells the router how to match the URL.

{ path: '', redirectTo: '/url', pathMatch: 'full' }, // redirect to `first-component`

# **Communicating with backend services using HTTP**

Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and access other back-end services. Angular provides a simplified client HTTP API for Angular applications, the [HttpClient](https://angular.io/api/common/http/HttpClient) service class in @angular/common/[http](https://angular.io/api/common/http).

The HTTP client service offers the following major features.

* The ability to request [typed response objects](https://angular.io/guide/http#typed-response).
* Streamlined [error handling](https://angular.io/guide/http#error-handling).
* [Testability](https://angular.io/guide/http#testing-requests) features.
* Request and response [interception](https://angular.io/guide/http#intercepting-requests-and-responses).

## **Setup for server communication**

Before you can use [HttpClient](https://angular.io/api/common/http/HttpClient), you need to import the Angular [HttpClientModule](https://angular.io/api/common/http/HttpClientModule). Most apps do so in the root AppModule.

import { HttpClientModule} from '@angular/common/http';

imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule,

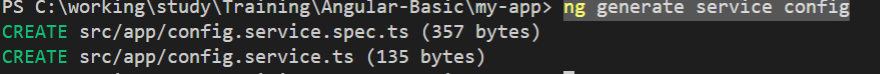
    ReactiveFormsModule,

    HttpClientModule

  ],

Create Service –

ng generate service config



You can then inject the [HttpClient](https://angular.io/api/common/http/HttpClient) service as a dependency of an application class, as shown in the following ConfigService example.

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

import { HttpClient } from '@angular/common/http';

@Injectable({

  providedIn: 'root'

})

export class ConfigService {

  constructor(private http: HttpClient) { }

}

The [HttpClient](https://angular.io/api/common/http/HttpClient) service makes use of [observables](https://angular.io/guide/glossary#observable) for all transactions. You must import the RxJS observable and operator symbols that appear in the example snippets. These ConfigService imports are typical.

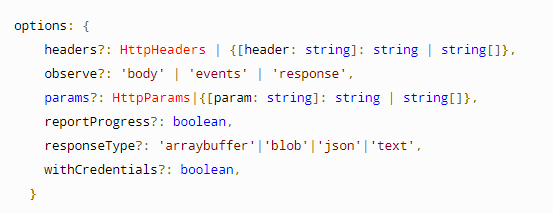
import { Observable, throwError } from 'rxjs';

import { catchError, retry } from 'rxjs/operators';

## **Requesting data from a server**

Use the [HTTPClient.get()](https://angular.io/api/common/http/HttpClient" \l "get) method to fetch data from a server. The asynchronous method sends an HTTP request, and returns an Observable that emits the requested data when the response is received. The return type varies based on the observe and responseType values that you pass to the call.

The get() method takes two arguments; the endpoint URL from which to fetch, and an options object that you can use to configure the request.



Important options include the *observe* and *responseType* properties.

* The *observe* option specifies how much of the response to return.
* The *responseType* option specifies the format in which to return data.

getConfig() { return this.http.get(this.configUrl); }

Final config.service.ts

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

import { HttpClient } from '@angular/common/http';

import { Observable, throwError } from 'rxjs';

import { catchError, retry } from 'rxjs/operators';

@Injectable({

  providedIn: 'root'

})

export class ConfigService {

  configUrl = 'https://reqres.in/api/users?page=2';

  constructor(private http: HttpClient) { }

  getConfig() {

    return this.http.get(this.configUrl);

  }

}

Next is to create a component where we will use this service

ng generate component user

in user component just include the service and read the data as below

@Component({

  selector: 'app-user',

  templateUrl: './user.component.html',

  styleUrls: ['./user.component.css'],

  providers: [ConfigService]

})

export class UserComponent implements OnInit {

  users = [];

  error: any;

  constructor(private configService: ConfigService) { }

  ngOnInit(): void {

    this.configService.getConfig()

      .subscribe(

        (data: any) => this.users = data.data , // success path

        error => this.error = error // error path

      );

  }

}