**How does an Angular application work?**

Every Angular app consists of a file named **angular.json**. This file will contain all the configurations of the app. While building the app, the builder looks at this file to find the entry point of the application. Following is an image of the angular.json file:

"build": {

"builder": "@angular-devkit/build-angular:browser",

"options": {

"outputPath": "dist/angular-starter",

"index": "src/index.html",

"main": "src/main.ts",

"polyfills": "src/polyfills.ts",

"tsConfig": "tsconfig.app.json",

"aot": false,

"assets": [

"src/favicon.ico",

"src/assets"

],

"styles": [

"./node\_modules/@angular/material/prebuilt-themes/deeppurple-amber.css",

"src/style.css"

]

}

}

Inside the build section, the main property of the options object defines the entry point of the application which in this case is **main.ts**.  
The main.ts file creates a browser environment for the application to run, and, along with this, it also calls a function called **bootstrapModule**, which bootstraps the application. These two steps are performed in the following order inside the main.ts file:

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

platformBrowserDynamic().bootstrapModule(AppModule)

In the above line of code, **AppModule** is getting bootstrapped.  
The AppModule is declared in the app.module.ts file. This module contains declarations of all the components.  
Below is an example of app.module.ts file:

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

entryComponents: [],

bootstrap: [AppComponent]

})

export class AppModule { }

As one can see in the above file, **AppComponent** is getting bootstrapped.  
This component is defined in **app.component.ts** file. This file interacts with the webpage and serves data to it.  
Below is an example of app.component.ts file:

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'angular';

}

Each component is declared with three properties:  
1. **Selector** - used for accessing the component  
2. **Template/TemplateURL** - contains HTML of the component  
3. **StylesURL** - contains component-specific stylesheets  
  
After this, Angular calls the **index.html** file. This file consequently calls the root component that is **app-root**. The root component is defined in **app.component.ts**.  
This is how the index.html file looks:

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Angular</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

</head>

<body>

<app-root></app-root>

</body>

</html>

The HTML template of the root component is displayed inside the <app-root> tags.  
  
This is how every angular application works.

**What are some of the advantages of Angular over other frameworks?**

 **Features that are provided out of the box -** Angular provides a number of built-in features like,routing, state management, rxjs library and http servicesstraight out of the box. This means that one does not need tolook for the above stated features separately. They are allprovided with angular.

 **Declarative UI -**Angular uses HTML to render the UI of an application. HTML isa declarative language and is much easier to use than JavaScript.

 **Long-term Google support -**Google announced Long-term support for Angular. This means that Google plans to stick with Angular and further scale up its ecosystem.

### List out differences between AngularJS and Angular

#### Architecture

AngularJS uses MVC or Model-View-Controller architecture, where the Model contains the business logic, Controller processes information and View shows the information present in the Model.  
Angular replaces controllers with Components. Components are nothing but directives with a predefined template.

#### Language

AngularJS uses JavaScript language, which is a dynamically typed language.  
Angular uses TypeScript language, which is a statically typed language and is a superset of JavaScript. By using statically typed language, Angular provides better performance while developing larger applications.

#### Mobile Support

AngularJS does not provide mobile support.  
Angular is supported by all popular mobile browsers.

#### Structure

While developing larger applications, the process of maintaining code becomes tedious in the case of AngularJS.  
In the case of Angular, it is easier to maintain code for larger applications as it provides a better structure.

#### Expression Syntax

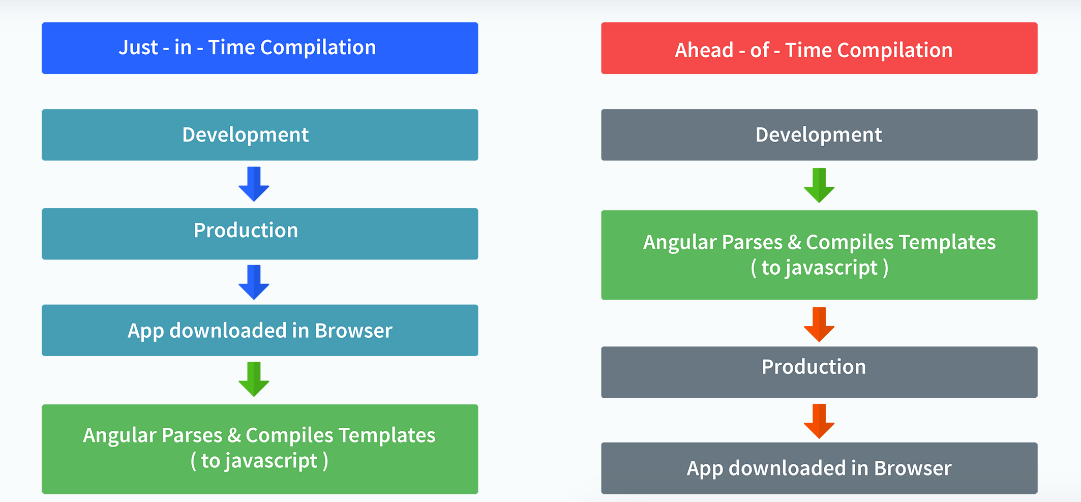
While developing an AngularJS application, a developer needs to remember the correct ng-directive for binding an event, or a property. Whereas in Angular, property binding is done using "[ ]" attribute and event binding is done using "( )" attribute.

**What is AOT compilation? What are the advantages of AOT?**

Every Angular application consists of components and templates which the browser cannot understand. Therefore, all the Angular applications need to be compiled first before running inside the browser.  
  
Angular provides two types of compilation:

 JIT(Just-in-Time) compilation

 AOT(Ahead-of-Time) compilation



In JIT compilation, the application compiles inside the browser during runtime.  
Whereas in the AOT compilation, the application compiles during the build time.  
  
The advantages of using AOT compilation are:

 Since the application compiles before running inside the browser, the browser loads the executable code and renders the application immediately, which leads to **faster rendering**.

 In AOT compilation, the compiler sends the external HTML and CSS files along with the application, eliminating separate AJAX requests for those source files, which leads to **fewer ajax requests**.

 Developers can detect and handle errors during the building phase, which helps in **minimizing errors**.

 The AOT compiler adds HTML and templates into the JS files before they run inside the browser. Due to this, there are no extra HTML files to be read, which provide **better security** to the application.

By default, angular builds and serves the application using JIT compiler:

ng build  
ng serve

For using AOT compiler following changes should be made:

ng build --aot  
ng serve --aot

What is a decorator in Angular 8?

Decorators are **a design pattern that is used to separate modification or decoration of a class without modifying the original source code**

**Explain Components, Modules and Services in Angular**

For better understanding, I would like you to create an Angular application by running the following inside the command terminal:

ng new angularApp

The above command will create an angular application in the directory.  
Next, let's move on to understand Components, Modules, and Services.  
  
**Components**  
In Angular, components are the basic building blocks, which control a part of the UI for any application.  
A component is defined using the **@Component** decorator. Every component consists of three parts, the template which loads the view for the component, a stylesheet which defines the look and feel for the component, and a class that contains the business logic for the component.  
For creating a component, inside the command terminal, navigate to the directory of the application created, and run the following command:

ng generate component test

Or

ng g c test

**Modules**  
A module is a place where we can group components, directives, services, and pipes. Module decides whether the components, directives, etc can be used by other modules, by exporting or hiding these elements. Every module is defined with a @NgModule decorator.

By default, modules are of two types:

 Root Module

 Feature ModuleEvery application can have only one root module whereas, it can have one or more feature modules.  
A root module imports **BrowserModule**, whereas a feature module imports **CommonModule**.  
  
In the application that we created before, one can see that the root module is defined inside **app.module.ts** and this is how it looks:

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

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

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

import { TestComponent } from './test/text.component';

@NgModule({

declarations: [

AppComponent,

TestComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

We can see in the above image that the component we created earlier is already imported in the declarations array.  
  
To create a feature module, run the following command:

ng g m test-module

The module is created inside the src/app/test-module/test-module.module.ts file:

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

import { CommonModule } from '@angular/common';

@NgModule({

declarations: [],

imports: [

CommonModule

]

})

export class TestModuleModule { }

As one can see, **CommonModule** is imported since this is a feature module.

**Services** Services are objects which get instantiated only once during the lifetime of an application. The main objective of a service is to share data, functions with different components of an Angular application.  
A service is defined using a **@Injectable** decorator. A function defined inside a service can be invoked from any component or directive.  
  
To create a service, run the following command:

ng g s test-service

The service will be created inside src/app/test-service.service.ts:

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

@Injectable({

providedIn: 'root'

})

export class TestServiceService {

constructor() { }

}

Any method/function defined inside the TestServiceService class can be directly used inside any component by just importing the service.

# **NgModules**

**NgModules** configure the injector and the compiler and help organize related things together.

An NgModule is a class marked by the @[NgModule](https://angular.io/api/core/NgModule) decorator. @[NgModule](https://angular.io/api/core/NgModule) takes a metadata object that describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes, making some of them public, through the exports property, so that external components can use them. @[NgModule](https://angular.io/api/core/NgModule) can also add service providers to the application dependency injectors.

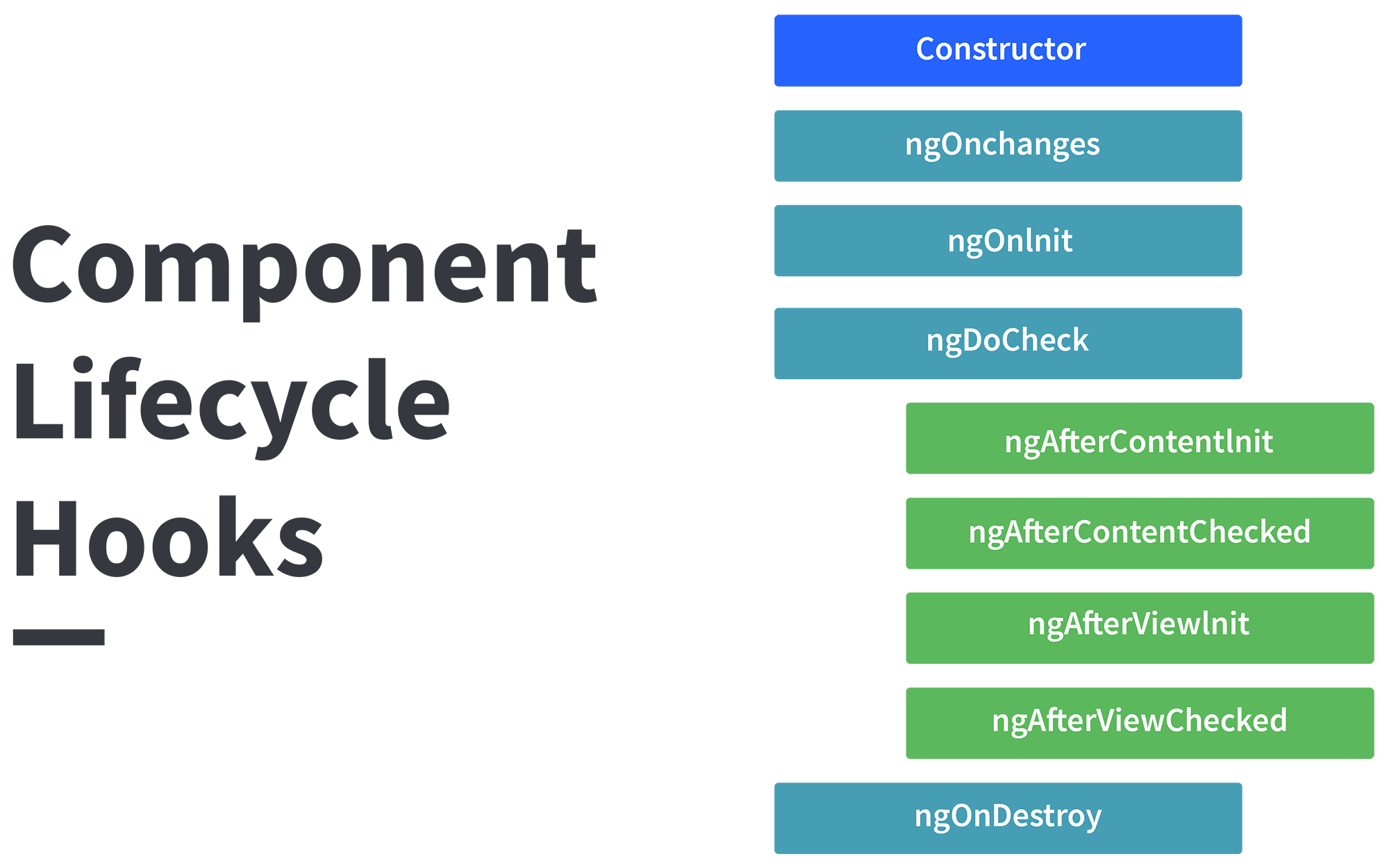
NgModule metadata does the following:

* Declares which components, directives, and pipes belong to the module
* Makes some of those components, directives, and pipes public so that other module's component templates can use them
* Imports other modules with the components, directives, and pipes that components in the current module need
* Provides services that other application components can use

Every Angular application has at least one module, the root module. You [bootstrap](https://angular.io/guide/bootstrapping) that module to launch the application.

### What are lifecycle hooks in Angular? Explain a few lifecycle hooks.

Every component in Angular has a lifecycle, different phases it goes through from the time of creation to the time it's destroyed. Angular provides **hooks** to tap into these phases and trigger changes at specific phases in a lifecycle.



**ngOnChanges( )** This hook/method is called before **ngOnInit** and whenever one or more input properties of the component changes.  
This method/hook receives a SimpleChanges object which contains the previous and current values of the property.  
  
**ngOnInit( )** This hook gets called once, after the **ngOnChanges** hook.  
It initializes the component and sets the input properties of the component.  
  
**ngDoCheck( )** It gets called after **ngOnChanges** and **ngOnInit** and is used to detect and act on changes that cannot be detected by Angular.  
We can implement our change detection algorithm in this hook. **ngAfterContentInit( )** It gets called after the first **ngDoCheck** hook. This hook responds after the content gets projected inside the component.  
  
**ngAfterContentChecked( )** It gets called after **ngAfterContentInit** and every subsequent **ngDoCheck**. It responds after the projected content is checked.  
  
**ngAfterViewInit( )** It responds after a component's view, or a child component's view is initialized.  
  
**ngAfterViewChecked( )** It gets called after **ngAfterViewInit**, and it responds after the component's view, or the child component's view is checked.  
  
**ngOnDestroy( )** It gets called just before Angular destroys the component. This hook can be used to clean up the code and detach event handlers.  
  
Let’s understand how to use **ngOnInit** hook, since it’s the most oftenly used hook. If one has to process lot of data during component creation, it’s better to do it inside **ngOnInit** hook rather than the constructor:

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

@Component({

selector: 'app-test',

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

styleUrls: ['./test.component.css']

})

export class TestComponent implements OnInit {

constructor() { }

ngOnInit() {

this.processData();

}

processData(){

// Do something..

}

}

As you can see we have imported OnInit but we have used **ngOnInit** function. This principle should be used with the rest of the hooks as well

### How are observables different from promises?

Observable in Angular is **a feature that provides support for delivering messages between different parts of your single-page application**. This feature is frequently used in Angular because it is responsible for handling multiple values, asynchronous programming in Javascript, and also event handling processesThe first difference is that an Observable is **lazy** whereas a Promise is **eager**.

What Is Promise in Angular? Promises in Angular **provide an easy way to execute asynchronous functions that use callbacks, while emitting and completing (resolving or rejecting) one value at a time**. When using an Angular Promise, you are enabled to emit a single event from the API

|  |  |
| --- | --- |
| Promise | Observable |
| Emits a single value | Emits multiple values over a period of time |
| Not Lazy | Lazy. An observable is not called until we subscribe to the observable |
| Cannot be cancelled | Can be cancelled by using the unsubscribe() method |
|  | Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc. |

Consider the following Observable:

const observable = rxjs.Observable.create(observer => {

console.log('Text inside an observable');

observer.next('Hello world!');

observer.complete();

});

console.log('Before subscribing an Observable');

observable.subscribe((message)=> console.log(message));

When you run the above Observable, you can see messages being displayed in the following order:

Before subscribing an Observable  
Text inside an observable  
Hello world!

As you can see, observables are lazy. Observable runs only when someone subscribes to them hence, the message “Before subscribing…” is displayed ahead of the message inside the observable.  
  
Now let’s consider a Promise:

const promise = new Promise((resolve, reject) => {

console.log('Text inside promise');

resolve('Hello world!');

});

console.log('Before calling then method on Promise');

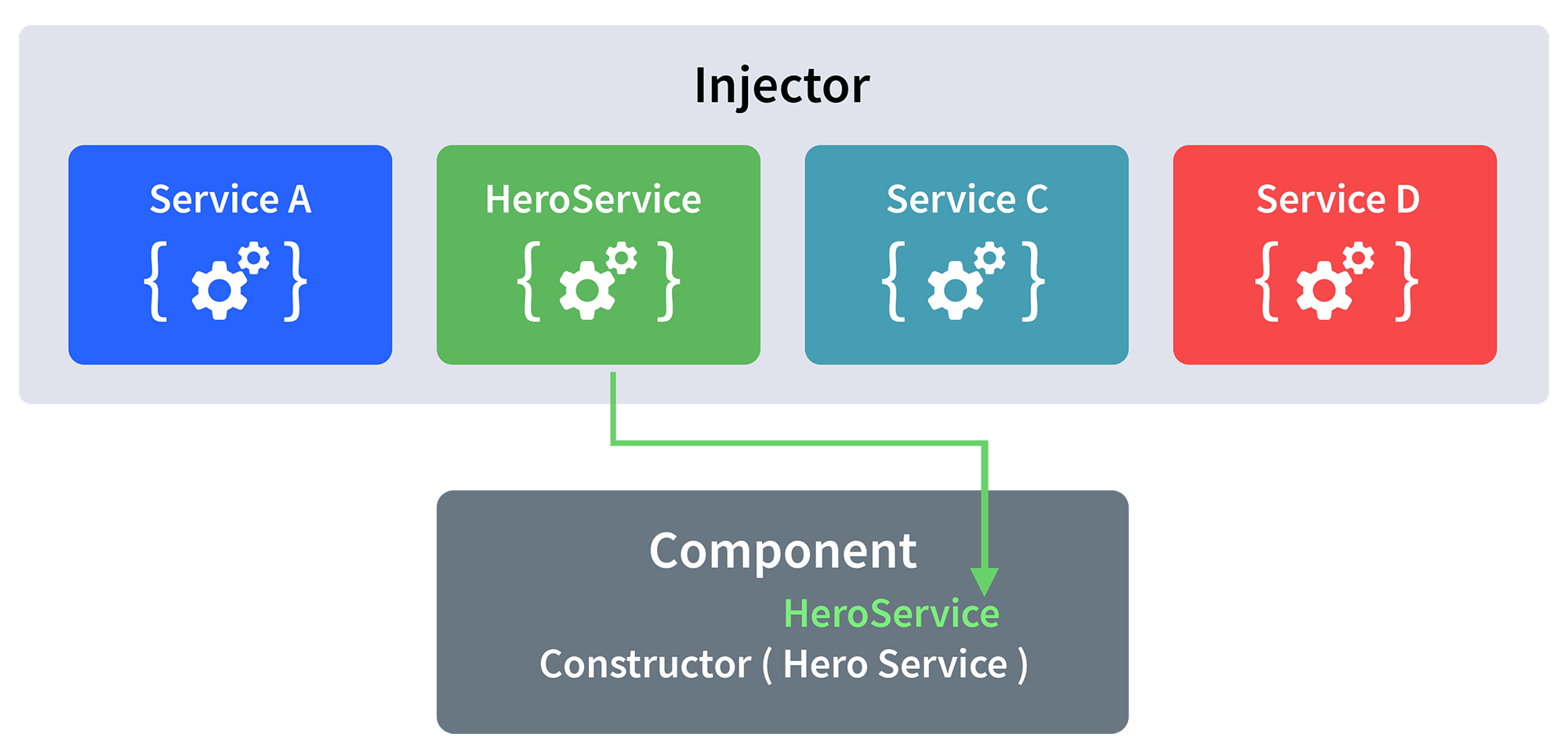
greetingPoster.then(message => console.log(message));

Running the above promise, the messages will be displayed in the following order:

Text inside promise  
Before calling then method on Promise  
Hello world!

### Explain the concept of Dependency Injection?

Dependency injection is an application design pattern which is implemented by Angular.  
It also forms one of the core concepts of Angular.  
  
**So what is dependency injection in simple terms?**  
Let’s break it down, dependencies in angular are nothing but services which have a functionality. Functionality of a service, can be needed by various components and directives in an application. Angular provides a smooth mechanism by which we can inject these dependencies in our components and directives.  
So basically, we are just making dependencies which are injectable across all components of an application.



Let’s understand how DI (Dependency Injection) works:  
  
Consider the following service, which can be generated using:

ng g service test

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

@Injectable({

providedIn: 'root'

})

export class TestService {

importantValue:number = 42;

constructor() { }

returnImportantValue(){

return this.importantValue;

}

}

As one can notice, we can create injectable dependencies by adding the **@Injectable** decorator to a class.  
  
We inject the above dependency inside the following component:

import { TestService } from './../test.service';

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

@Component({

selector: 'app-test',

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

styleUrls: ['./test.component.css']

})

export class TestComponent implements OnInit {

value:number;

constructor(private testService:TestService) { }

ngOnInit() {

this.value = this.testService.returnImportantValue();

}

}

One can see we have imported our TestService at the top of the page. Then, we have created an instance inside the constructor of the component and implemented the **returnImportantValue** function of the service.  
  
From the above example, we can observe how angular provides a smooth way to inject dependencies in any component

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses the double curly braces {{ and }} as delimiters.

To illustrate how interpolation works, consider an Angular component that contains a currentCustomer variable:

content\_copycurrentCustomer = 'Maria';

Use interpolation to display the value of this variable in the corresponding component template:

content\_copy<h3>Current customer: {{ currentCustomer }}</h3>

Angular replaces currentCustomer with the string value of the corresponding component property. In this case, the value is Maria

## **Binding to events**

To bind to an event you use the Angular event binding syntax. This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right.

Create the following example; the target event name is click and the template statement is onSave().

Event binding syntax

content\_copy<button (click)="onSave()">Save</button>

The event binding listens for the button's click events and calls the component's onSave() method whenever a click occurs.

### Determining an event target

To determine an event target, Angular checks if the name of the target event matches an event property of a known directive.

Create the following example: (Angular checks to see if myClick is an event on the custom ClickDirective)

src/app/app.component.html

content\_copy<h4>myClick is an event on the custom ClickDirective:</h4>

<button type="button" (myClick)="clickMessage=$event" clickable>click with myClick</button>

{{clickMessage}}

If the target event name, myClick fails to match an output property of ClickDirective, Angular will instead bind to the myClick event on the underlying DOM element.

# **Property binding**

Property binding in Angular helps you set values for properties of HTML elements or directives. Use property binding to do things such as toggle button functionality, set paths programmatically, and share values between components.

See the [live example](https://angular.io/generated/live-examples/property-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/property-binding/property-binding.zip) for a working example containing the code snippets in this guide.

## **Prerequisites**

* [Basics of components](https://angular.io/guide/architecture-components)
* [Basics of templates](https://angular.io/guide/glossary#template)
* [Binding syntax](https://angular.io/guide/binding-syntax)

## **Understanding the flow of data**

Property binding moves a value in one direction, from a component's property into a target element property.

For more information on listening for events, see [Event binding](https://angular.io/guide/event-binding).

To read a target element property or call one of its methods, see the API reference for [ViewChild](https://angular.io/api/core/ViewChild) and [ContentChild](https://angular.io/api/core/ContentChild).

## **Binding to a property**

To bind to an element's property, enclose it in square brackets, [], which identifies the property as a target property.

A target property is the DOM property to which you want to assign a value.

To assign a value to a target property for the image element's src property, type the following code:

src/app/app.component.html

content\_copy<img alt="item" [src]="itemImageUrl">

In most cases, the target name is the name of a property, even when it appears to be the name of an attribute.

In this example, src is the name of the <img> element property.

The brackets, [], cause Angular to evaluate the right-hand side of the assignment as a dynamic expression.

Without the brackets, Angular treats the right-hand side as a string literal and sets the property to that static value.

To assign a string to a property, type the following code:

src/app.component.html

content\_copy<app-item-detail childItem="parentItem"></app-item-detail>

Omitting the brackets renders the string parentItem, not the value of parentItem

# **Two-way binding**

Two-way binding gives components in your application a way to share data. Use two-way binding to listen for events and update values simultaneously between parent and child components.

See the [live example](https://angular.io/generated/live-examples/two-way-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/two-way-binding/two-way-binding.zip) for a working example containing the code snippets in this guide.

## **Prerequisites**

To get the most out of two-way binding, you should have a basic understanding of the following concepts:

* [Property binding](https://angular.io/guide/property-binding)
* [Event binding](https://angular.io/guide/event-binding)
* [Inputs and Outputs](https://angular.io/guide/inputs-outputs)

Two-way binding combines property binding with event binding:

| BINDINGS | DETAILS |
| --- | --- |
| [Property binding](https://angular.io/guide/property-binding) | Sets a specific element property. |
| [Event binding](https://angular.io/guide/event-binding) | Listens for an element change event. |

## **Adding two-way data binding**

Angular's two-way binding syntax is a combination of square brackets and parentheses, [()]. The [()] syntax combines the brackets of property binding, [], with the parentheses of event binding, (), as follows.

src/app/app.component.html

content\_copy<app-sizer [(size)]="fontSizePx"></app-sizer>

## **How two-way binding works**

For two-way data binding to work, the @[Output](https://angular.io/api/core/Output)() property must use the pattern, inputChange, where input is the name of the @[Input](https://angular.io/api/core/Input)() property. For example, if the @[Input](https://angular.io/api/core/Input)() property is size, the @[Output](https://angular.io/api/core/Output)() property must be sizeChange.

The following sizerComponent has a size value property and a sizeChange event. The size property is an @[Input](https://angular.io/api/core/Input)(), so data can flow into the sizerComponent. The sizeChange event is an @[Output](https://angular.io/api/core/Output)(), which lets data flow out of the sizerComponent to the parent component.

Next, there are two methods, dec() to decrease the font size and inc() to increase the font size. These two methods use resize() to change the value of the size property within min/max value constraints, and to emit an event that conveys the new size value.

src/app/sizer.component.ts

content\_copyexport class SizerComponent {

@[Input](https://angular.io/api/core/Input)() size!: number | string;

@[Output](https://angular.io/api/core/Output)() sizeChange = new [EventEmitter](https://angular.io/api/core/EventEmitter)<number>();

dec() { this.resize(-1); }

inc() { this.resize(+1); }

resize(delta: number) {

this.size = Math.min(40, Math.max(8, +this.size + delta));

this.sizeChange.emit(this.size);

}

}

The sizerComponent template has two buttons that each bind the click event to the inc() and dec() methods. When the user clicks one of the buttons, the sizerComponent calls the corresponding method. Both methods, inc() and dec(), call the resize() method with a +1 or -1, which in turn raises the sizeChange event with the new size value.

src/app/sizer.component.html

content\_copy<div>

<button type="button" (click)="dec()" title="smaller">-</button>

<button type="button" (click)="inc()" title="bigger">+</button>

<span [style.font-size.px]="size">FontSize: {{size}}px</span>

</div>

In the AppComponent template, fontSizePx is two-way bound to the SizerComponent.

src/app/app.component.html

content\_copy<app-sizer [(size)]="fontSizePx"></app-sizer>

<div [style.font-size.px]="fontSizePx">Resizable Text</div>

In the AppComponent, fontSizePx establishes the initial SizerComponent.size value by setting the value to 16.

src/app/app.component.ts

content\_copyfontSizePx = 16;

Clicking the buttons updates the AppComponent.fontSizePx. The revised AppComponent.fontSizePx value updates the style binding, which makes the displayed text bigger or smaller.

The two-way binding syntax is shorthand for a combination of property binding and event binding. The SizerComponent binding as separate property binding and event binding is as follows.

src/app/app.component.html (expanded)

content\_copy<app-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></app-sizer>

The $event variable contains the data of the SizerComponent.sizeChange event. Angular assigns the $event value to the AppComponent.fontSizePx when the user clicks the buttons.

TWO-WAY BINDING IN

# **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.

Try this [Reactive Forms live-example](https://angular.io/generated/live-examples/reactive-forms/stackblitz.html) / [download example](https://angular.io/generated/zips/reactive-forms/reactive-forms.zip).

## **Prerequisites**

Before going further into reactive forms, you should have a basic understanding of the following:

* [TypeScript](https://www.typescriptlang.org/) programming
* Angular application-design fundamentals, as described in [Angular Concepts](https://angular.io/guide/architecture)
* The form-design concepts that are presented in [Introduction to Forms](https://angular.io/guide/forms-overview)

## **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 synchronous access to the data model, immutability with observable operators, and change tracking through observable streams.

Template-driven forms let direct access 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 application. 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.

You can then display the form by adding the component to the template.

The following examples show how to add a single form control. In the example, the user enters their name into an input field, captures that input value, and displays the current value of the form control element.

| ACTION | DETAILS |
| --- | --- |
| 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.src/app/app.module.ts (excerpt)  content\_copyimport { [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) | Use the [CLI command](https://angular.io/cli) ng generate to generate a component in your project to host the control.src/app/name-editor/name-editor.component.ts  content\_copyimport { [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-name-editor',  templateUrl: './name-editor.component.html',  styleUrls: ['./name-editor.component.css']  })  export class NameEditorComponent {  name = new [FormControl](https://angular.io/api/forms/FormControl)('');  }  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. |
| Register the control in the template | After you create the control in the component class, you must associate it with a form control element in the template. Update the template with the form control using the formControl binding provided by [FormControlDirective](https://angular.io/api/forms/FormControlDirective), which is also included in the [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule).src/app/name-editor/name-editor.component.html  content\_copy<label for="name">Name: </label>  <input id="name" type="text" [formControl]="name">   * For a summary of the classes and directives provided by [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule), see the following [Reactive forms API](https://angular.io/guide/reactive-forms#reactive-forms-api) section * For complete syntax details of these classes and directives, see the API reference documentation for the [Forms package](https://angular.io/api/forms)   Using the template binding syntax, the form control is now registered to the name input element in the template. The form control and DOM element communicate with each other: the view reflects changes in the model, and the model reflects changes in the view. |
| Display the component | The form control assigned to name is displayed when the component is added to a template.src/app/app.component.html (name editor)  content\_copy<app-name-editor></app-name-editor>  Name Editor, which has a name label and an input so the user can enter a name |

### 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.

src/app/name-editor/name-editor.component.html (control value)

content\_copy<p>Value: {{ name.value }}</p>

The displayed value changes as you update the form control element.

Reactive forms provide access to information about a given control through properties and methods provided with each instance. These properties and methods of the underlying [AbstractControl](https://angular.io/api/forms/AbstractControl" \o "API reference) class are used to control form state and determine when to display messages when handling [input validation](https://angular.io/guide/reactive-forms#basic-form-validation).

Read about other [FormControl](https://angular.io/api/forms/FormControl) properties and methods in the [API Reference](https://angular.io/api/forms/FormControl).

### Replacing a form control value

Reactive forms have methods to change a control's value programmatically, which gives you the flexibility to update the value without user interaction. A form control instance provides a setValue() method that updates the value of the form control and validates the structure of the value provided against the control's structure. For example, when retrieving form data from a backend API or service, use the setValue() method to update the control to its new value, replacing the old value entirely.

The following example adds a method to the component class to update the value of the control to Nancy using the setValue() method.

src/app/name-editor/name-editor.component.ts (update value)

content\_copyupdateName() {

this.name.setValue('Nancy');

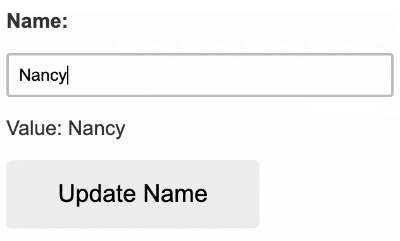
}

Update the template with a button to simulate a name update. When you click the **Update Name** button, the value entered in the form control element is reflected as its current value.

src/app/name-editor/name-editor.component.html (update value)

content\_copy<button type="button" (click)="updateName()">Update Name</button>

The form model is the source of truth for the control, so when you click the button, the value of the input is changed within the component class, overriding its current value.



**NOTE**:  
In this example, you're using a single control. When using the setValue() method with a [form group](https://angular.io/guide/reactive-forms#grouping-form-controls) or [form array](https://angular.io/guide/reactive-forms#creating-dynamic-forms) instance, the value needs to match the structure of the group or array.

## **Grouping form controls**

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

| FORM GROUPS | DETAILS |
| --- | --- |
| 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. |
| 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). |

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.

content\_copyng generate component ProfileEditor

src/app/profile-editor/profile-editor.component.ts (imports)

content\_copyimport { [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.

| ACTION | DETAILS |
| --- | --- |
| Create a [FormGroup](https://angular.io/api/forms/FormGroup) instance | Create a property in the component class named profileForm and set the property to a new form group instance. To initialize the form group, provide the constructor with an object of named keys mapped to their control. For the profile form, add two form control instances with the names firstName and lastName.src/app/profile-editor/profile-editor.component.ts (form group)  content\_copyimport { [Component](https://angular.io/api/core/Component) } from '@angular/core';  import { [FormGroup](https://angular.io/api/forms/FormGroup), [FormControl](https://angular.io/api/forms/FormControl) } from '@angular/forms';  @[Component](https://angular.io/api/core/Component)({  selector: 'app-profile-editor',  templateUrl: './profile-editor.component.html',  styleUrls: ['./profile-editor.component.css']  })  export class ProfileEditorComponent {  profileForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({  firstName: new [FormControl](https://angular.io/api/forms/FormControl)(''),  lastName: new [FormControl](https://angular.io/api/forms/FormControl)(''),  });  }  The individual form controls are now collected within a group. A [FormGroup](https://angular.io/api/forms/FormGroup) instance provides its model value as an object reduced from the values of each control in the group. A form group instance has the same properties (such as value and untouched) and methods (such as setValue()) as a form control instance. |
| Associate the [FormGroup](https://angular.io/api/forms/FormGroup) model and view | A form group tracks the status and changes for each of its controls, so if one of the controls changes, the parent control also emits a new status or value change. The model for the group is maintained from its members. After you define the model, you must update the template to reflect the model in the view.src/app/profile-editor/profile-editor.component.html (template form group)  content\_copy<form [formGroup]="profileForm">  <label for="first-name">First Name: </label>  <input id="first-name" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="firstName">  <label for="last-name">Last Name: </label>  <input id="last-name" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="lastName">  </form>  **NOTE**: Just as a form group contains a group of controls, the profileForm [FormGroup](https://angular.io/api/forms/FormGroup) is bound to the form element with the [FormGroup](https://angular.io/api/forms/FormGroup) directive, creating a communication layer between the model and the form containing the inputs.  The [formControlName](https://angular.io/api/forms/FormControlName) input provided by the [FormControlName](https://angular.io/api/forms/FormControlName) directive binds each individual input to the form control defined in [FormGroup](https://angular.io/api/forms/FormGroup). The form controls communicate with their respective elements. They also communicate changes to the form group instance, which provides the source of truth for the model value. |
| 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.src/app/profile-editor/profile-editor.component.html (submit event)  content\_copy<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">  The onSubmit() method in the ProfileEditor component captures the current value of profileForm. Use [EventEmitter](https://angular.io/api/core/EventEmitter) to keep the form encapsulated and to provide the form value outside the component. The following example uses console.warn to log a message to the browser console.src/app/profile-editor/profile-editor.component.ts (submit method)  content\_copyonSubmit() {  // TODO: Use [EventEmitter](https://angular.io/api/core/EventEmitter) with form value  console.warn(this.profileForm.value);  }  The submit event is emitted by the form tag using the built-in DOM event. You trigger the event by clicking a button with submit type. This lets the user press the **Enter** key to submit the completed form. Use a button element to add a button to the bottom of the form to trigger the form submission.src/app/profile-editor/profile-editor.component.html (submit button)  content\_copy<p>Complete the form to enable button.</p>  <button type="submit" [disabled]="!profileForm.valid">Submit</button>  **NOTE**: The button in the preceding snippet also has a disabled binding attached to it to disable the button when profileForm is invalid. You aren't performing any validation yet, so the button is always enabled. Basic form validation is covered in the [Validating form input](https://angular.io/guide/reactive-forms#basic-form-validation) section. |
| Display the component | To display the ProfileEditor component that contains the form, add it to a component template.src/app/app.component.html (profile editor)  content\_copy<app-profile-editor></app-profile-editor>  ProfileEditor lets you manage the form control instances for the firstName and lastName controls within the form group instance.  Profile Editor with labels and inputs for first and last name as well as a submit button |

### Creating nested form groups

# **Building a template-driven form**

This tutorial shows you how to create a template-driven form whose control elements are bound to data properties, with input validation to maintain data integrity and styling to improve the user experience.

Template-driven forms use [two-way data binding](https://angular.io/guide/architecture-components#data-binding) to update the data model in the component as changes are made in the template and vice versa.

Angular supports two design approaches for interactive forms. You can build forms by writing templates using Angular [template syntax and directives](https://angular.io/guide/glossary#template) with the form-specific directives and techniques described in this tutorial, or you can use a reactive (or model-driven) approach to build forms.

Template-driven forms are suitable for small or simple forms, while reactive forms are more scalable and suitable for complex forms. For a comparison of the two approaches, see [Introduction to Forms](https://angular.io/guide/forms-overview)

You can build almost any kind of form with an Angular template —login forms, contact forms, and pretty much any business form. You can lay out the controls creatively and bind them to the data in your object model. You can specify validation rules and display validation errors, conditionally enable or disable specific controls, trigger built-in visual feedback, and much more.

This tutorial shows you how to build a form from scratch, using a simplified sample form like the one from the [Tour of Heroes tutorial](https://angular.io/tutorial) to illustrate the techniques.

Run or download the example app: [live example](https://angular.io/generated/live-examples/forms/stackblitz.html) / [download example](https://angular.io/generated/zips/forms/forms.zip).

## **Objectives**

This tutorial teaches you how to do the following:

* Build an Angular form with a component and template
* Use [ngModel](https://angular.io/api/forms/NgModel) to create two-way data bindings for reading and writing input-control values
* Provide visual feedback using special CSS classes that track the state of the controls
* Display validation errors to users and enable or disable form controls based on the form status
* Share information across HTML elements using [template reference variables](https://angular.io/guide/template-reference-variables)

## **Prerequisites**

Before going further into template-driven forms, you should have a basic understanding of the following.

* [TypeScript](https://www.typescriptlang.org/) and HTML5 programming
* Angular app-design fundamentals, as described in [Angular Concepts](https://angular.io/guide/architecture)
* The basics of [Angular template syntax](https://angular.io/guide/template-syntax)
* The form-design concepts that are presented in [Introduction to Forms](https://angular.io/guide/forms-overview)

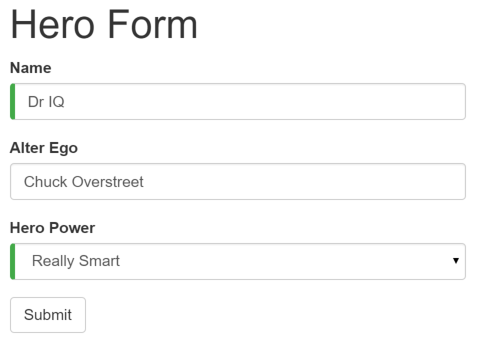
## **Build a template-driven form**

Template-driven forms rely on directives defined in the [FormsModule](https://angular.io/api/forms/FormsModule).

| DIRECTIVES | DETAILS |
| --- | --- |
| [NgModel](https://angular.io/api/forms/NgModel) | Reconciles value changes in the attached form element with changes in the data model, allowing you to respond to user input with input validation and error handling. |
| [NgForm](https://angular.io/api/forms/NgForm) | Creates a top-level [FormGroup](https://angular.io/api/forms/FormGroup) instance and binds it to a <form> element to track aggregated form value and validation status. As soon as you import [FormsModule](https://angular.io/api/forms/FormsModule), this directive becomes active by default on all <form> tags. You don't need to add a special selector. |
| [NgModelGroup](https://angular.io/api/forms/NgModelGroup) | Creates and binds a [FormGroup](https://angular.io/api/forms/FormGroup) instance to a DOM element. |

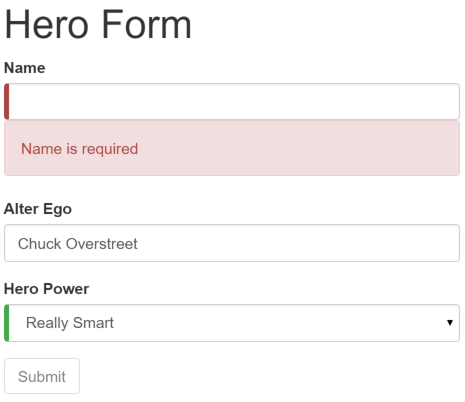
### The sample application

The sample form in this guide is used by the Hero Employment Agency to maintain personal information about heroes. Every hero needs a job. This form helps the agency match the right hero with the right crisis.



The form highlights some design features that make it easier to use. For instance, the two required fields have a green bar on the left to make them easy to spot. These fields have initial values, so the form is valid and the **Submit** button is enabled.

As you work with this form, you will learn how to include validation logic, how to customize the presentation with standard CSS, and how to handle error conditions to ensure valid input. If the user deletes the hero name, for example, the form becomes invalid. The application detects the changed status, and displays a validation error in an attention-grabbing style. In addition, the **Submit** button is disabled, and the "required" bar to the left of the input control changes from green to red.



### Step overview

In the course of this tutorial, you bind a sample form to data and handle user input using the following steps.

1. Build the basic form.
   * Define a sample data model
   * Include required infrastructure such as the [FormsModule](https://angular.io/api/forms/FormsModule)
2. Bind form controls to data properties using the [ngModel](https://angular.io/api/forms/NgModel) directive and two-way data-binding syntax.
   * Examine how [ngModel](https://angular.io/api/forms/NgModel) reports control states using CSS classes
   * Name controls to make them accessible to [ngModel](https://angular.io/api/forms/NgModel)
3. Track input validity and control status using [ngModel](https://angular.io/api/forms/NgModel).
   * Add custom CSS to provide visual feedback on the status
   * Show and hide validation-error messages
4. Respond to a native HTML button-click event by adding to the model data.
5. Handle form submission using the [ngSubmit](https://angular.io/api/forms/NgForm" \l "properties) output property of the form.
   * Disable the **Submit** button until the form is valid
   * After submit, swap out the finished form for different content on the page