[Skip to content](/l)

* [Safari Home](/home/)
* [Recommended](/r/)
* [Queue](/s/)
* HYPERLINK \l ""
* HYPERLINK \l ""
* [History](/history/)
* [Topics](/topics)
* [Tutorials](/tutorials/)
* [Offers & Deals](/l)
* [Newsletters](/oreilly-newsletters/)
* [Highlights](/u/0011N000019ujUhQAI/)
* [Settings](/u/)
* [Support](https://www.safaribooksonline.com/public/support)
* [Sign Out](/accounts/logout/)
* HYPERLINK "/u/"HYPERLINK "/subscribe/"
* HYPERLINK "https://www.safaribooksonline.com/public/support"
* HYPERLINK "/accounts/logout/"

[**Table of Contents for Angular UI Development with PrimeNG**](/l)

* [Search in book...](/l)

Top of Form 1



* [Bottom of Form 1](Bottom%20of%20Form%201)
* [Toggle Font Controls](/l)
* HYPERLINK \l ""
* HYPERLINK "https://twitter.com/share?url=https://www.safaribooksonline.com/library/view/angular-ui-development/9781788299572/f1d254d5-922e-415e-8f6d-7c415db81013.xhtml&text=Angular%20UI%20Development%20with%20PrimeNG&via=safari"
* HYPERLINK "https://www.facebook.com/sharer/sharer.php?u=https://www.safaribooksonline.com/library/view/angular-ui-development/9781788299572/f1d254d5-922e-415e-8f6d-7c415db81013.xhtml"
* HYPERLINK "https://plus.google.com/share?url=https://www.safaribooksonline.com/library/view/angular-ui-development/9781788299572/f1d254d5-922e-415e-8f6d-7c415db81013.xhtml"
* HYPERLINK "mailto:?subject=Safari: Angular%20cheat%20sheet%20-%20overview%20of%20key%20concepts&body=https://www.safaribooksonline.com/library/view/angular-ui-development/9781788299572/f1d254d5-922e-415e-8f6d-7c415db81013.xhtml%0D%0Afrom Angular%20UI%20Development%20with%20PrimeNG%0D%0A"
* [Prev Previous Chapter](/library/view/angular-ui-development/9781788299572/f1d254d5-922e-415e-8f6d-7c415db81013.xhtml)
* [Angular cheat sheet - overview of key concepts](Angular%20cheat%20sheet%20-%20overview%20of%20key%20concepts)
* [Next Next Chapter](/library/view/angular-ui-development/9781788299572/e22ff469-9a18-4564-8979-52d192295715.xhtml)
* [Templates and bindings](Templates%20and%20bindings)
* **Components, services, and dependency injection**
* Normally, you write Angular applications by composing HTML templates with the Angular-specific markup and component classes to manage those templates. A **component** is simply a TypeScript class annotated with **@Component**. The **@Component** decorator is used to define the associated metadata. It expects an object with the following most used properties:
* **selector**: This is the name of the HTML tag representing this component
* **template**: This is an inline-defined template with HTML/Angular markup for the view
* **templateUrl**: This is the path to an external file where the template resides
* **styles**: An inline-defined styles to be applied to this component's view
* **styleUrls**: An array of paths to external files with styles to be applied to this component's view
* **providers**: An array of providers available to this component and its children
* **exportAs**: This is the name under which the component instance is exported in a template
* **changeDetection**: This is the change detection strategy used by this component
* **encapsulation**: This is the style encapsulation strategy used by this component
* A component class interacts with the view through an API of properties and methods. Component classes should delegate complex tasks to services where the business logic resides. **Services** are just classes that Angular instantiates and then injects into components. If you register services at the root component level, they act as singletons and share data across multiple components. In the next section, *Angular modularity and lifecycle hooks*, we will see how to register services. The following example demonstrates how to use components and services. We will write a service class **ProductService** and then specify an argument of type **ProductService** in the constructor of **ProductComponent**. Angular will automatically inject that service into the component:
* import {Injectable, Component} from '@angular/core';  
    
  @Injectable()  
  export class ProductService {  
   products: Product[];  
    
   getProducts(): Array<Product> {  
   // retrieve products from somewhere...  
   return products;  
   }  
  }  
    
  @Component({  
   selector: 'product-count',  
   template: `<h2 class="count">Found {{products.length}} products</h2>`,  
   styles: [`  
   h2.count {  
   height: 80px;  
   width: 400px;  
   }  
   `]  
  })  
  export default class ProductComponent {  
   products: Product[] = [];  
    
   constructor(productService: ProductService) {  
   this.products = productService.getProducts();  
   }  
  }
* Notice that we applied the **@Injectable()** decorator to the service class. This is necessary for emitting metadata that Angular needs to inject other dependencies into this service. Using **@Injectable** is a good programming style even if you don't inject other services into your service.
* It is good to know what an item in the **providers** array looks like. An item is an object with the **provide** property (symbol used for dependency injection) and one of the three properties **useClass**, **useFactory**, or **useValue** that provide implementation details:
* {provide: MyService, useClass: MyMockService}  
  {provide: MyService, useFactory: () => {return new MyMockService()}}  
  {provide: MyValue, useValue: 50}

**Templates and bindings**

A template tells Angular how to render the component's view. Templates are HTML snippets with the specific Angular's template syntax, such as interpolation, property, attribute, and event bindings, built-in directives, and pipes to mention just a few. We will give you a quick overview of the template syntax starting with interpolation. **Interpolation** is used to evaluate expressions in double curly braces. The evaluated expression is then converted to a string. The expression can contain any mathematical calculations, component's properties and methods, and many more:

<p>Selected car is {{currentCar.model}}</p>

Angular evaluates template expressions after every change detection cycle. Change detection cycles are triggered by many asynchronous activities such as HTTP responses, key and mouse events, and many more. The next fundamental template syntax is related to various bindings. *Property binding* sets an element property to a component property value. The element property is defined in square brackets:

<img [src]="imageUrl">  
<button [disabled]="formValid">Submit</button>

Here, **imageUrl** and **formValid** are a component's properties. Note that this is a *one-way* binding because the data flow occurs in one direction, from the component's properties into target element properties. *Attribute binding* allows us to set an attribute. This kind of binding is used when there is no element property to bind. The attribute binding uses square brackets too. The attribute name itself is prefixed with **attr.**, for example, consider ARIA attributes for web accessibility:

<button [attr.aria-expanded]="expanded" [attr.aria-controls]="controls">  
 Click me  
</button>

User interactions result in a data flow from an element to a component. In Angular, we can listen for certain key, mouse, and touch events by means of *event binding*. The event binding syntax consists of a target event name within parentheses on the left and a quoted template statement on the right. In particular, you can call a component's method. In the next code snippet, the **onSave()** method is called on a click:

<button (click)="onSave()">Save</button>

The method (generally template statement) gets a parameter--an event object named **$event**. For native HTML elements and events, **$event** is a DOM event object:

<input [value]="name" (input)="name=$event.target.value">

Two-way binding is possible as well. The **[(value)]** syntax combines the brackets of property binding with the parentheses of event binding. Angular's directive **NgModel** is best suited for the two-way binding on native or custom input elements. Consider the following sample:

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

Is equivalent to:

<input [value]="username" (input)="username=$event.target.value">

Two-way binding in a nutshell: a property gets displayed and updated at the same time when the user makes changes. A *template reference variable* is another example of handy template syntax. You can declare a variable with the hash symbol (**#**) on any DOM element and reference this variable anywhere in the template. The next example shows the **username** variable declared on an **input** element. This reference variable is consumed on a button--it is used to get an input value for the **onclick** handler:

<input #username>  
<button (click)="submit(username.value)">Ok</button>

A template reference variable can also be set to a directive. A typical example is the **NgForm** directive which provides useful details about the **form** elements. You can, for example, disable the submit button if the form is not valid (required fields are not filled in and so on):

<form #someForm="ngForm">  
 <input name="name" required [(ngModel)]="name">  
 ...  
 <button type="submit" [disabled]="!someForm.form.valid">Ok</button>  
</form>

Last but not least, the *pipe* operator (**|**). It is used for the transformation of the expression's result. The pipe operator passes the result of an expression on the left to a pipe function on the right. For example, the pipe **date** formats JavaScript **Date** object according to the specified format (<https://angular.io/docs/ts/latest/api/common/index/DatePipe-pipe.html>):

Release date: {{releaseDate | date: 'longDate'}}  
// Output: "August 30, 2017"

Multiple chained pipes can be applied as well.

**Built-in directives**

Angular has a lot of built-in directives: **ngIf**, **ngFor**, **ngSwitch**, **ngClass**, and **ngStyle**. The first three directives are so called *structural directives*, which are used to transform the DOM's structure. Structural directives start with an asterisk (**\***). The last two directives manipulate the CSS classes and styles dynamically. Let's explain the directives in the examples.

The **ngIf** directive adds and removes elements in the DOM, based on the Boolean result of an expression. In the next code snippet, **<h2>ngIf</h2>** is removed when the **show** property evaluates to **false** and gets recreated otherwise:

<div \*ngIf="show">  
 <h2>ngIf</h2>  
</div>

Angular 4 has introduced a new **else** clause with the reference name for a template defined by **ng-template**. The content within **ng-template** is shown when the **ngIf** condition evaluates to **false**:

<div \*ngIf="showAngular; else showWorld">  
 Hello Angular  
</div>  
<ng-template #showWorld>  
 Hello World  
</ng-template>

**ngFor** outputs a list of elements by iterating over an array. In the next code snippet, we iterate over the **people** array and store each item in a template variable called **person**. This variable can be then accessed within the template:

<ui>  
 <li \*ngFor="let person of people">  
 {{person.name}}  
 </li>  
</ui>

**ngSwitch** conditionally swaps the contents dependent on condition. In the next code snippet, **ngSwitch** is bound to the **choice** property. If **ngSwitchCase** matches the value of this property, the corresponding HTML element is displayed. If no matching exists, the element associated with **ngSwitchDefault** is displayed:

<div [ngSwitch]="choice">  
 <h2 \*ngSwitchCase="'one'">One</h3>  
 <h2 \*ngSwitchCase="'two'">Two</h3>  
 <h2 \*ngSwitchDefault>Many</h3>  
</div>

**ngClass** adds and removes CSS classes on an element. The directive should receive an object with class names as keys and expressions as values that evaluate to **true** or **false**. If the value is **true**, the associated class is added to the element. Otherwise, if **false**, the class is removed from the element:

<div [ngClass]="{selected: isSelected, disabled: isDisabled}">

**ngStyle** adds and removes inline styles on an element. The directive should receive an object with style names as keys and expressions as values that evaluate to style values. A key can have an optional **.<unit>** suffix (for example, **top.px**):

<div [ngStyle]="{'color': 'red', 'font-weight': 'bold', 'border-top': borderTop}">

In order to be able to use built-in directives in templates, you have to import **CommonModule** from **@angular/common** and add it to the root module of your application. Angular's modules are explained in the next chapter.

**Communication between components**

Components can communicate with each other in a loosely coupled manner. There are various ways Angular's components can share data, including the following:

* Passing data from parent to child using **@Input()**
* Passing data from child to parent using **@Output()**
* Using services for data sharing
* Calling **ViewChild**, **ViewChildren**, **ContentChild**, and **ContentChildren**
* Interacting with the child component using a local variable

We will only describe the first three ways. A component can declare input and output properties. To pass the data from a parent to a child component, the parent binds the values to the input properties of the child. The child's input property should be decorated with **@Input()**. Let's create **TodoChildComponent**:

@Component({  
 selector: 'todo-child',  
 template: `<h2>{{todo.title}}</h2>`  
})  
export class TodoChildComponent {  
 @Input() todo: Todo;  
}

Now, the parent component can use **todo-child** in its template and bind the parent's **todo** object to the child's **todo** property. The child's property is exposed as usual in square brackets:

<todo-child [todo]="todo"></todo-child>

If a component needs to pass the data to its parent, it emits custom events via the output property. The parent can create a listener to a particular component's event. Let's see that in action. The child component **ConfirmationChildComponent** exposes an **EventEmitter** property decorated with **@Output()** to emit events when the user clicks on buttons:

@Component({  
 selector: 'confirmation-child',  
 template: `  
 <button (click)="accept(true)">Ok</button>  
 <button (click)="accept(false)">Cancel</button>  
 `  
})  
export class ConfirmationChildComponent {  
 @Output() onAccept = new EventEmitter<boolean>();  
  
 accept(accepted: boolean) {  
 this.onAccept.emit(accepted);  
 }  
}

The parent subscribes an event handler to that event property and reacts to the emitted event:

@Component({  
 selector: 'confirmation-parent',  
 template: `  
 Accepted: {{accepted}}  
 <confirmation-child (onAccept)="onAccept($event)"></confirmation-child>  
 `  
})  
export class ConfirmationParentComponent {  
 accepted: boolean = false;  
  
 onAccept(accepted: boolean) {  
 this.accepted = accepted;  
 }  
}

A bi-directional communication is possible via services. Angular leverages RxJS library (<https://github.com/Reactive-Extensions/RxJS>) for asynchronous and event-based communication between several parts of an application as well as between an application and remote backend. The key concepts in the asynchronous and event-based communication are **Observer** and **Observable**. They provide a generalized mechanism for push-based notification, also known as the observer design pattern. **Observable** represents an object that sends notifications, and **Observer** represents an object that receives them.

Angular implements this design pattern everywhere. For example, Angular's **Http** service returns an **Observable** object:

constructor(private http: Http) {}  
  
getCars(): Obvervable<Car[]> {  
 return this.http.get("../data/cars.json")  
 .map(response => response.json().data as Car[]);  
}

In case of the inter-component communication, an instance of the **Subject** class can be used. This class inherits both **Observable** and **Observer**. That means it acts as a message bus. Let's implement **TodoService** that allows us to emit and receive **Todo** objects:

@Injectable()  
export class TodoService {  
 private subject = new Subject();  
  
 toggle(todo: Todo) {  
 this.subject.next(todo);  
 }  
  
 subscribe(onNext, onError, onComplete) {  
 this.subject.subscribe(onNext, onError, onComplete);  
 }  
}

Components can use this service in the following way:

export class TodoComponent {  
 constructor(private todosService: TodosService) {}  
   
 toggle(todo: Todo) {  
 this.todosService.toggle(todo);  
 }  
}  
  
export class TodosComponent {  
 constructor(private todosService: TodosService) {  
 todosService.subscribe(  
 function(todo: Todo) { // TodoComponent sent todo object },  
 function(e: Error) { // error occurs },  
 function() { // completed }  
 );  
 }  
}

**Forms**

Forms are the main building blocks in every web application. Angular offers two approaches to build forms: *template-driven forms* and *reactive forms*. This section gives you a short overview of template-driven forms.

Reactive forms are suitable when you need to create dynamic forms programmatically in the component's class. Refer to the official Angular documentation to learn reactive forms (<https://angular.io/docs/ts/latest/guide/reactive-forms.html>).

We already mentioned two directives: **NgForm** and **NgModel**. The first directive creates a **FormGroup** instance and binds it to a form in order to track aggregate form value and validation status. The second one creates a **FormControl** instance and binds it to the corresponding **form** element. The **FormControl** instance tracks the value and the status of the **form** element. Each input element should have a **name** property that is required to register the **FormControl** by the **FormGroup** under the name you assigned to the **name** attribute. How to deal with this tracked data? You can export the **NgForm** and **NgModel** directives into local template variables such as **#f="ngForm"** and **#i="ngModel"**, respectively. Here, **f** and **i** are local template variables that give you access to the value and status of **FormGroup** and **FormControl**, respectively. This is possible because the properties from **FormGroup** and **FormControl** are duplicated on the directives themselves. With this knowledge in hand, you can now check if the whole form or a particular **form** element:

* Is valid (**valid** and **invalid** properties)
* Has been visited (**touched** and **untouched** properties)
* Has some changed value (**dirty** and **pristine** properties)

The next example illustrates the basic concept:

<form #f="ngForm" (ngSubmit)="onSubmit(f)" novalidate>  
 <label for="name">Name</label>  
 <input type="text" id=name" name="name" required  
 [(ngModel)]="name" #i="ngModel">   
 <div [hidden]="i.valid || i.pristine">  
 Name is required  
 </div>  
 <button>Submit</button>  
</form>  
  
// Output values and states  
Input value: {{i.value}}  
Is input valid? {{i.valid}}  
Input visited? {{i.touched}}  
Input value changed? {{i.dirty}}  
Form input values: {{f.value | json}}  
Is form valid? {{f.valid}}  
Form visited? {{f.touched}}  
Form input values changed? {{f.dirty}}

The **NgModel** directive also updates the corresponding **form** element with specific CSS classes that reflect the element's state. The following classes are added/removed dependent on the current state:

|  |  |  |
| --- | --- | --- |
| **State** | **Class if true** | **Class if false** |
| Element has been visited | **ng-touched** | **ng-untouched** |
| Element's value has changed | **ng-dirty** | **ng-pristine** |
| Element's value is valid | **ng-valid** | **ng-invalid** |

This is handy for styling. For example, in case of validation errors, you can set red borders around input elements:

input.ng-dirty.ng-invalid {  
 border: solid 1px red;  
}

**Routing**

Angular's **router** module allows you to configure navigation in a single page application without a full page reload. The router can display different views (compiled component templates) within a special tag called **<router-outlet>**. During navigation, one view will be replaced by another one. A simple routing configuration looks as follows:

const router: Routes = [  
 {path: '', redirectTo: 'home', pathMatch: 'full'},  
 {path: 'home', component: HomeComponent},  
 {path: 'books', component: BooksComponent}  
];

When you navigate to the web context root, you will be redirected to **/home**. As a reaction to that, the view of the **HomeComponent** will be displayed in **<router-outlet>**. It is obvious that a direct navigation to **/home** displays the same view. A navigation to **/books** displays the view of **BooksComponent**. Such router configuration should be converted to an Angular's module by **RouterModule.forRoot**:

const routes: ModuleWithProviders = RouterModule.forRoot(router);

This is then imported in a root module class. In addition to the root module, an Angular application can consist of a lot of feature or lazy-loaded modules. Such separate modules can have their own router configurations which should be converted to Angular's modules with **RouterModule.forChild(router)**. The next section, *Angular modularity and lifecycle hooks,* discusses modules in detail. Angular offers two strategies for implementing client-side navigation:

* **HashLocationStrategy**: This strategy adds a hash sign (**#**) to the base URL. Everything after this sign represents a hash fragment of the browser's URL. The hash fragment identifies the route. For example, [**http://somehost.de:8080/#/books**](http://somehost.de:8080/#/books). Changing the route doesn't cause a server-side request. Instead, the Angular application navigates to a new route and view. This strategy works with all browsers.
* **PathLocationStrategy**: This strategy is based on the **History API** and only works in browsers that support HTML5. This is the default location strategy.

The details are to be mentioned here. If you want to use the **HashLocationStrategy**, you have to import two classes, **LocationStrategy** and **HashLocationStrategy** from **'@angular/common'** and configure providers as follows:

providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}]

Providers are described in the next section, *Angular modularity and lifecycle hooks.* The **PathLocationStrategy** class requires a configuration of the base URL for the entire application. The best practice is to import **APP\_BASE\_HREF** constant from **'@angular/common'** and use it as a provider in order to configure the base URL:

providers: [{provide: APP\_BASE\_HREF, useValue: '/'}]

How to trigger a navigation? You can achieve that in two ways, either by a link with a **routerLink** property, which specifies an array consisting of route (path) and optional parameters:

<a [routerLink]="['/']">Home</a>  
<a [routerLink]="['/books']">Books</a>  
  
<router-outlet></router-outlet>

Or programmatically, by invoking the **navigate** method on Angular's **Router** service:

import {Router} from '@angular/router';  
  
...  
  
export class HomeComponent {  
   
 constructor(private router: Router) { }  
  
 gotoBooks() {  
 this.router.navigate(['/books']);  
 }  
}

You can also pass parameters to a route. Placeholders for parameters start with a colon (**:**):

const router: Routes = [  
 ...  
 {path: 'books/:id', component: BookComponent}  
];

Now, when navigating to a book with real parameters, for example, programmatically as **this.router.navigate(['/books/2'])**, the real parameter can be accessed by **ActivatedRoute**:

import {ActivatedRoute} from '@angular/router';  
  
...  
  
export class BooksComponent {  
 book: string;  
   
 constructor(private route: ActivatedRoute) {  
 this.book = route.snapshot.params['id'];  
 }  
}

The router outlet can be named as well:

<router-outlet name="author"></router-outlet>

The associated configuration should contain the **outlet** property with the name of the router outlet:

{path: 'author', component: AuthorComponent, outlet: 'author'}