Angular is a javascript framework which allows you to create reactive single page applications.

Single Page Application: One html file, javascript change content [DOM] during run time.

**Versioning :**

Angular 1 is called angular js

Angular 2 and above are called angular applications, there are major changes in 1 and 2, but there are minor changes from 2, 4, 6 ,7 , 8.

Angular 3 is not released.

**Project Setup:**

1. First install node js, its provide the **npm** node package manager, it contains tool to download dependencies, so its must be installed, simply download and install.
2. Now need to install angular cli by npm : **npm install -g @angular/cli@latest**
3. Now create new application by command **ng new app\_name**
4. To run this go to project directory and run **ng serve**
5. npm install @angular/cli

To install modules for existing project

1. Now we look project structure
   1. Online sell - project Name
      1. .**settings**
      2. .**vscode**
      3. .**node\_modules** - contains all modules which are needed for our app. Like **bootstrap**
      4. **Src** - contains all developer src code [component, services, models]
         1. Component - all new components
         2. Services
         3. Model
         4. App-routing-module.js : This is used for routing, when routing needed, routing is the way by which we can change component by click.
         5. App.component.css : CSS file for main component
         6. App.component.html : HTML file for main component
         7. App.component.ts : Type script file for main component.
         8. App.module.ts : configuration file to load modules.
      5. **Angular.json** - just like microservice **application.properties**
      6. **package.json** - just like **pom.xml**
      7. **Index.html**
      8. **main.ts**

**App.module.ts :**

If you create a component or service, to use another component or service, you need to register here so that others can be aware of that. So any thing which you needed by outside as well, you must need to import over here.

You need to import and put that one in import section. Like

1. Import {ramjicomponent} from “./component/admin/admin.component”
2. Imports: [ramjicomponent]

**How project Boot:**

1. Main.ts file is responsible for that, it contains bootstrap module, which is AppModule
2. Now the system will look appmodule file which contains all the services and components registered. First component is app.component which is created by default.
3. It will then load index,html, which have reference app.component
4. Now app.component loaded and the rest of the things will work as you develop.

**Creating a component:**

**Ng g c component\_name** : **Component will be registered in app.module.ts.**

**g** for generate

**c** for component:

It will generate three files:

1. **ts file** : it's basically a class file like java which binds css file and html file and you can write down dynamic code here like you write in java.

Imports ;

@component({

Selector : name of component,

templateURL : Name of HTML file with path [“”],

styleURL : Name of css file with path [“”],

Styles : [`your css if you want ‘]

})

Export class class\_name implement onInit {

}

1. **css file**
2. **html file**

**Create service** :

Ng g s service\_name : s for service name and it will also register in app.module.ts

Imports ;

@injectable({

providedIn: ‘root’

})

Export class class\_name implement onInit {

}

Browsers only understand javascript: so a typescript file would be converted to a js file by angular cli.

**Imp** : In place of templateUrl we can use template and use our own html content.

## **Workspace configuration files**

| WORKSPACE CONFIG FILES | PURPOSE |
| --- | --- |
| .editorconfig | Configuration for code editors. See [EditorConfig](https://editorconfig.org/). |
| .gitignore | Specifies intentionally untracked files that [Git](https://git-scm.com/) should ignore. |
| README.md | Introductory documentation for the root app. |
| angular.json | CLI configuration defaults for all projects in the workspace, including configuration options for build, serve, and test tools that the CLI uses, such as [TSLint](https://palantir.github.io/tslint/), [Karma](https://karma-runner.github.io/), and [Protractor](https://www.protractortest.org/). For details, see [Angular Workspace Configuration](https://angular.io/guide/workspace-config). |
| package.json | Configures [npm package dependencies](https://angular.io/guide/npm-packages) that are available to all projects in the workspace. See [npm documentation](https://docs.npmjs.com/files/package.json) for the specific format and contents of this file. |
| package-lock.json | Provides version information for all packages installed into node\_modules by the npm client. See [npm documentation](https://docs.npmjs.com/files/package-lock.json) for details. If you use the yarn client, this file will be [yarn.lock](https://yarnpkg.com/lang/en/docs/yarn-lock/) instead. |
| src/ | Source files for the root-level application project. |
| node\_modules/ | Provides [npm packages](https://angular.io/guide/npm-packages) to the entire workspace. Workspace-wide node\_modules dependencies are visible to all projects. |
| tsconfig.json | The base [TypeScript](https://www.typescriptlang.org/) configuration for projects in the workspace. All other configuration files inherit from this base file. For more information, see the [Configuration inheritance with extends](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html#configuration-inheritance-with-extends) section of the TypeScript documentation. |
| tslint.json | Default [TSLint](https://palantir.github.io/tslint/) configuration for projects in the workspace. |

**Data Binding**: How Data would be shown from ts file to htm file.

There are three ways:

1. String interpolation : {{ **variable\_name** }} : It would be string only.
2. Property binding : If want to bind with any attribute of any component. Like id of a button so

<button **[id]=variable\_name**> button\_Name</button>

1. Event binding : by clicking or some events like

<button **[id]=variable\_name (click)= functionName($event, id)**> button\_Name</button>

Imp : All events are the same but we need to remove on and need to put into parenthesis like.

onClick = (click)

onMouseClick = (mouseclick)

Using Service in to component or in Another service:

1. First register that service into app.module.ts
2. Now in constructor of that class you will use like

constructor(private object\_name : service\_name){

}

You can also create object like c = new C();

1. If you want to use locally that service then put provider: [service name], so this object class would be available in component only.
2. Angular use singleton pattern, so every where you will use the same object of that class.

Component1 (modelclass)------- service [Model class]-----------Component2(model class)

**Router:**

Router is the substitution of #. In # page would be reloaded but by router dom would be reloaded so page loaded problem resolve.

Steps :

1. Enable router in your project.
2. app.Router.module.ts file is there
3. Or in appmodules.ts, import route
4. Now create constant like : const routes: Routes = [

{“path\_name” : component: componentname}

]

Path name would be used in place of #.

1. Now in imports in app.module.ts import:[Routermodule.forRoot(routes)]
2. In component use like <router-outlet></router-outlet>
3. Add a navigation link ([routerLink](https://angular.io/api/router/RouterLink))

<a [routerLink](https://angular.io/api/router/RouterLink)="/heroes">Heroes</a>

We can also use child router as well.

1. Set Default path for RouterLink.

 { path: '', redirectTo: '/dashboard', pathMatch: 'full' },

**Observable:**

It is just a publisher subscriber pattern.

Publisher or observable will emit the data or you can say, it will publish data, but in Angular you need to use pull base. If the publisher will change something then you need to listen to that.

Publisher is basically a source of data, you need to create a stream and the subscriber or observer will listen to that.

There are three conditions which we need to handle.

1. Data
2. Error
3. Completion

**publisher.data.subscribe();**

.subscribe(data => {

Your operations;

},error =>{

Your operations

},

()=>{

Completion code

}

);

Angular Directive: -

### List heroes with \*[ngFor](https://angular.io/api/common/NgForOf) :- The [\*ngFor](https://angular.io/guide/built-in-directives#ngFor) is Angular's repeater directive. It repeats the host element for each element in a list.

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

Don't forget the asterisk (\*) in front of [ngFor](https://angular.io/api/common/NgForOf). It's a critical part of the syntax.

#### **hide empty details with \*ngIf**

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

The two components will have a parent/child relationship. The parent HeroesComponent will control the child HeroDetailComponent by sending it a new hero to display whenever the user selects a hero from the list

preceded by the @[Input](https://angular.io/api/core/Input)() decorator.

@[Input](https://angular.io/api/core/Input)() hero: Hero;

## **Why services**

Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service.

Services are a great way to share information among classes that *don't know each other*. You'll create a MessageService and inject it in two places.

1. Inject in HeroService, which uses the service to send a message.
2. Inject in MessagesComponent, which displays that message, and also displays the ID when the user clicks a hero.

## **Create the HeroService**

Using the Angular CLI, create a service called hero.

content\_copyng generate service hero

The command generates a skeleton HeroService class in src/app/hero.service.ts as follows:

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

content\_copyimport { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';

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

providedIn: 'root',

})

export class HeroService {

constructor() { }

}

Notice that the new service imports the Angular [Injectable](https://angular.io/api/core/Injectable) symbol and annotates the class with the @[Injectable](https://angular.io/api/core/Injectable)() decorator. This marks the class as one that participates in the dependency injection system.

The @[Injectable](https://angular.io/api/core/Injectable)() decorator accepts a metadata object for the service, the same way the @[Component](https://angular.io/api/core/Component)() decorator did for your component classes.

List of all the decorators in Angular and their usage.

Here’s the list of decorators available in Angular:

1. @NgModule
2. @Component
3. @Injectable
4. @Directive
5. @Pipe
6. @Input
7. @Output
8. @HostBinding
9. @HostListener
10. [@ContentChild](https://medium.com/u/d220ff7175e9?source=post_page-----71bdf4ad6976--------------------------------)
11. @ContentChildren
12. @ViewChild
13. @ViewChildren

Explanation of each Decorator:

1. @NgModule:

Defines a module that contains components, directives, pipes, and providers.

Usage:

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

@**[NgModule](https://angular.io/api/core/NgModule)**({  
declarations:[Component1, Component2],  
imports: [Module1, Module2],  
exports: [MyModule],  
providers: [Service1, Service2],  
bootstrap: [AppComponent]})  
class MyModule {}

2. @Component:

Declares that a class is a component and provides metadata about the component.

Usage:

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

[@Component](http://twitter.com/Component)({  
changeDetection?: ChangeDetectionStrategy  
viewProviders?: Provider[]  
moduleId?: string  
templateUrl?: string  
template?: string  
styleUrls?: string[]  
styles?: string[]  
animations?: any[]  
encapsulation?: ViewEncapsulation  
interpolation?: [string, string]  
entryComponents?: Array<Type<any> | any[]>  
preserveWhitespaces?: boolean  
  
// inherited from core/Directive  
selector?: string  
inputs?: string[]  
outputs?: string[]  
host?: {...}  
providers?: Provider[]  
exportAs?: string  
queries?: {...}  
})

## **class ComponentName{}**

3. @Injectable:

Declares that a class has dependencies that should be injected into the constructor when the dependency injector is creating an instance of this class.

Usage:

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

4. @Directive

Declares that a class is a directive and provides metadata about the directive.

Usage:

import { Directive } from ‘[@angular/core](http://twitter.com/angular/core)’; [@Directive](http://twitter.com/Directive)({  
selector?: string  
inputs?: string[]  
outputs?: string[]  
host?: {…}  
providers?: Provider[]  
exportAs?: string  
queries?: {…}  
})

5. @Pipe

Declares that a class is a pipe and provides metadata about the pipe.

Usage:

import { Pipe } from ‘[@angular/core](http://twitter.com/angular/core)’; [@Pipe](http://twitter.com/Pipe)({  
name: string  
pure?: boolean  
})

6. @Input

Declares an input property that you can update via property binding (example: <my-cmp [myProperty]="someExpression">).

import { Input } from ‘[@angular/core](http://twitter.com/angular/core)’;  
[@Input](http://twitter.com/Input)({  
bindingPropertyName?: string  
})

7. @OutPut

Declares an output property that fires events that you can subscribe to with an event binding (example: <my-cmp (myEvent)="doSomething()">).

import { Output } from ‘[@angular/core](http://twitter.com/angular/core)’;  
[@Output](http://twitter.com/Output)({  
bindingPropertyName?: string  
})

8. @HostBinding

Binds a host element property (here, the CSS class valid) to a directive/component property (isValid).

import { HostBinding } from ‘[@angular/core](http://twitter.com/angular/core)’;  
[@HostBinding](http://twitter.com/HostBinding)({  
hostPropertyName?: string  
})

9. @HostListener

Subscribes to a host element event (click) with a directive/component method (onClick), optionally passing an argument ($event).

import { HostListener } from ‘[@angular/core](http://twitter.com/angular/core)’;  
[@HostListener](http://twitter.com/HostListener)({  
eventName?: string  
args?: string[]  
})

10. @ContentChild

Binds the first result of the component content query (myPredicate) to a property (myChildComponent) of the class.

import { ContentChild } from ‘[@angular/core](http://twitter.com/angular/core)’;  
[@ContentChild](http://twitter.com/ContentChild)(Pane) pane: Pane;

11. @ContentChildren

Binds the results of the component content query (myPredicate) to a property (myChildComponents) of the class.

import { ContentChildren } from ‘[@angular/core](http://twitter.com/angular/core)’;  
[@ContentChildren](http://twitter.com/ContentChildren)(Pane) topLevelPanes: QueryList<Pane>;

12. @ViewChild

Binds the first result of the component view query (myPredicate) to a property (myChildComponent) of the class. Not available for directives.

import { ViewChild } from '[@angular/core](http://twitter.com/angular/core)';  
[@ViewChild](http://twitter.com/ViewChild)(Pane)

13. @ViewChildren

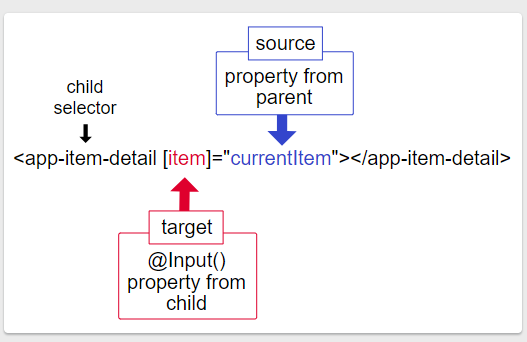
Binds the results of the component view query (myPredicate) to a property (myChildComponents) of the class. Not available for directives.

import { ViewChildren } from ‘[@angular/core](http://twitter.com/angular/core)’;  
[@ViewChildren](http://twitter.com/ViewChildren)(Pane) panes: QueryList<Pane>;

Sharing data between child and parent directives and components

A common pattern in Angular is sharing data between a parent component and one or more child components. You can implement this pattern by using the @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() directives.

@[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() give a child component a way to communicate with its parent component. @[Input](https://angular.io/api/core/Input)() allows a parent component to update data in the child component. Conversely, `@Output() allows the child to send data to a parent component.



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

<https://malcoded.com/posts/angular-fundamentals-modules/>

## What is an Angular Module?

Simply put, a module is just a class, just like [components](https://malcoded.com/posts/angular-fundamentals-components)and [services](https://malcoded.com/posts/angular-fundamentals-services).

The code in angular is generally organized in modules. You can think of modules like packages or bundles containing the required code for a specific use case.

The most prominent Module is the App-Module, because it comes with every new application, generated by the cli.

However, chances are that the App-Module is not the only module you have encountered so far. There are many other modules that come with angular out of the box.

Examples are the Http-Client-Module, contains a very useful Http-Client (surprise!) and the Forms-Module that contains UI components and directives to HTML-Forms.

As we have seen in the example above, we need to import a module first, before we can use it.

The root module of your application is the App-Module. This module imports other modules, which can import other modules them self.

Just like with components, the resulting structure is a module-tree.

### **Bootstrap**

Defines the root-component of the Application. Only use this in the AppModule.

### **Exports**

We define the components, directives or pipes we want to export here. That means, that our module is providing these to other modules when they get imported. Otherwise, these components stay module internal and cannot be accessed from the outside.

### **Declarations**

Inside of the declarations array, we define all the components, directives and pipes, that are declared and used inside this module. If a component (or directive or pipe) is not added to the declarations array and you use it in your module/application, angular will throw an error at runtime. Also, a component (or ... you got it) can only be declared in one module. If you want to use your component in multiple modules, you need to bundle that component into a separate module and import that in the module.

### **Imports**

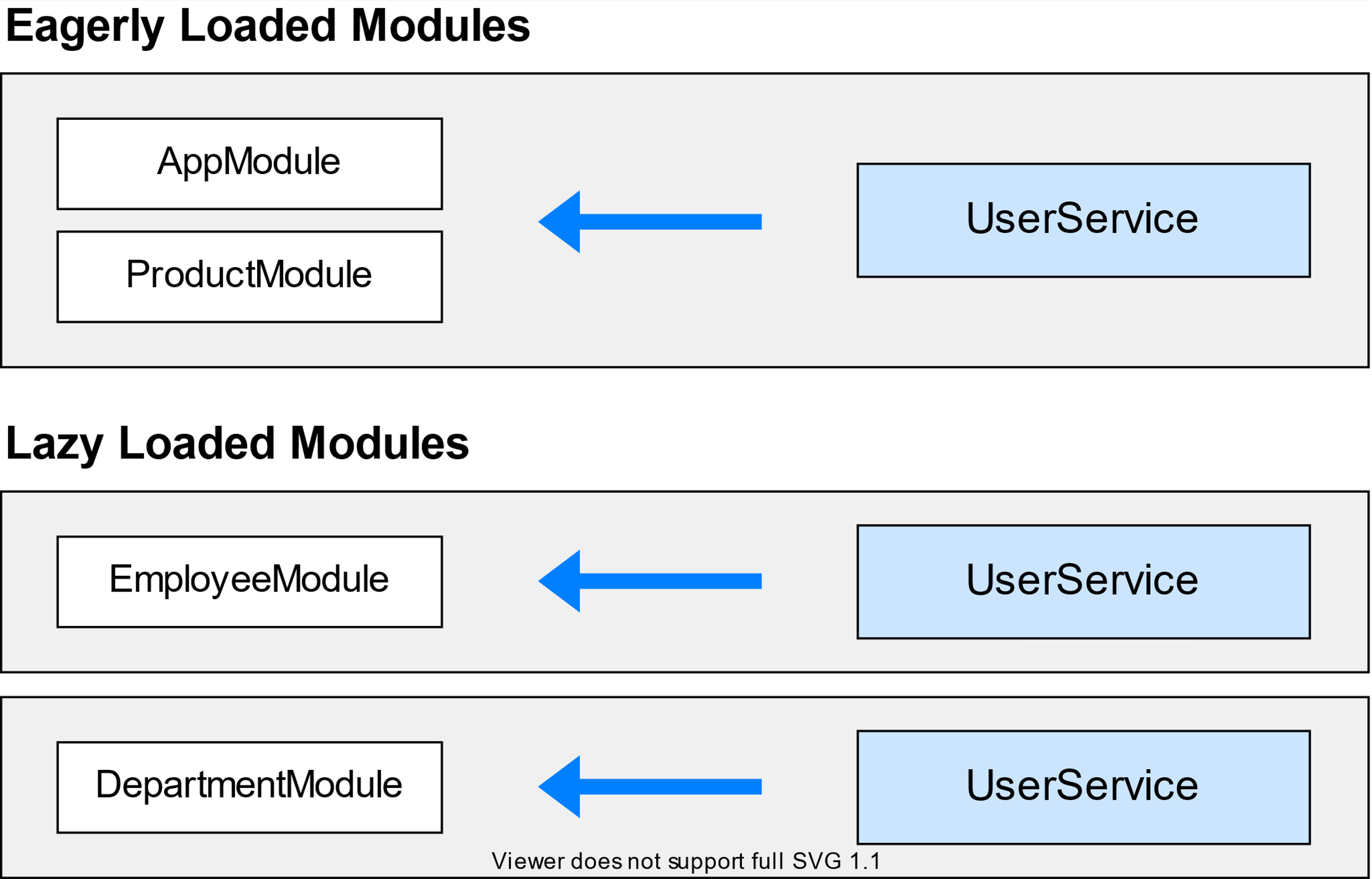
Speaking of importing... Your module can import as many sub-modules as you like. Don't have defined any custom modules yet? No problem, we will get to that. But even if you don't have any modules, you still need to import some angular modules. As I mentioned earlier, Angular is built with modularity in mind. While many features are contained in angular's core, some features are bundled into their own module. For example, if you want to use the [HttpClient](https://angular.io/api/common/http/HttpClient), you will need to import the [HttpClientModule](https://angular.io/api/common/http/HttpClientModule).

### **Providers**

We define any [@Injectables](https://malcoded.com/posts/angular-fundamentals-dependency-injection), required by the module, here. Any sub-components or modules can then get the same instance of that @Injectable via dependency injection. In the case of the AppModule, these @Injectables are application-scoped.

## Lazy-Loading Modules

It turns out, you can do more with modules, than just organizing your components. It is also possible to lazy-load modules. So what does that mean?  
Angular appears to be quite heavy in download size. Depending on your use case, that can be a big issue. Especially on mobile, it can take a while to download only the application. A way to reduce loading time, is to split your application into modules.



# **Providing dependencies in modules**

A provider is an instruction to the [Dependency Injection](https://angular.io/guide/dependency-injection) system on how to obtain a value for a dependency. Most of the time, these dependencies are services that you create and provide.

For the final sample app using the provider that this page describes, see the [live example](https://angular.io/generated/live-examples/providers/stackblitz.html) / [download example](https://angular.io/generated/zips/providers/providers.zip).

## **Providing a service**

If you already have an app that was created with the [Angular CLI](https://angular.io/cli), you can create a service using the [ng generate](https://angular.io/cli/generate) CLI command in the root project directory. Replace User with the name of your service.

content\_copyng generate service User

This command creates the following UserService skeleton:

src/app/user.service.ts

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

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

providedIn: 'root',

})

export class UserService {

}

You can now inject UserService anywhere in your application.

The service itself is a class that the CLI generated and that's decorated with @[Injectable](https://angular.io/api/core/Injectable)(). By default, this decorator has a providedIn property, which creates a provider for the service. In this case, providedIn: 'root' specifies that Angular should provide the service in the root injector.

## **Provider scope**

When you add a service provider to the root application injector, it’s available throughout the app. Additionally, these providers are also available to all the classes in the app as long they have the lookup token.

You should always provide your service in the root injector unless there is a case where you want the service to be available only if the consumer imports a particular @[NgModule](https://angular.io/api/core/NgModule).

## **providedIn and NgModules**

It's also possible to specify that a service should be provided in a particular @[NgModule](https://angular.io/api/core/NgModule). For example, if you don't want UserService to be available to applications unless they import a UserModule you've created, you can specify that the service should be provided in the module:

src/app/user.service.ts

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

import { UserModule } from './user.module';

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

providedIn: UserModule,

}

export class UserService {

}

The example above shows the preferred way to provide a service in a module. This method is preferred because it enables tree-shaking of the service if nothing injects it. If it's not possible to specify in the service which module should provide it, you can also declare a provider for the service within the module:

src/app/user.module.ts

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

import { UserService } from './user.service';

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

providers: [UserService],

})export class UserModule {}

Introduction to components and templates

<https://angular.io/guide/architecture-components>

A component controls a patch of screen called a [view](https://angular.io/guide/glossary#view). 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. The service is provided to the component through the dependency injection system.

## **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. The metadata for a component tells Angular where to get the major building blocks that it needs to create and present the component and its view. In particular, it associates a template with the component, either directly with inline code, or by reference. Together, the component and its template describe a view.

In addition to containing or pointing to the template, the @[Component](https://angular.io/api/core/Component) metadata configures, for example, how the component can be referenced in HTML and what services it requires.

@[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.

### **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.

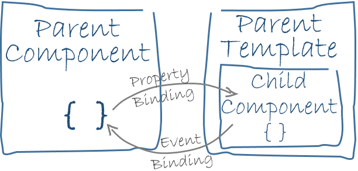
The following diagram shows the four forms of data binding markup. Each form has a direction: to the DOM, from the DOM, or both.



<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/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.
* src/app/hero-detail.component.html (ngModel)
* content\_copy<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name">
* In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.
* Angular processes all data bindings once 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.
* 

Feature modules

Feature modules are NgModules for the purpose of organizing code.

For the final sample app with a feature module that this page describes, see the [live example](https://angular.io/generated/live-examples/feature-modules/stackblitz.html) / [download example](https://angular.io/generated/zips/feature-modules/feature-modules.zip).

As your app grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features. With feature modules, you can keep code related to a specific functionality or feature separate from other code. Delineating areas of your app helps with collaboration between developers and teams, separating directives, and managing the size of the root module.

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

A feature module is an organizational best practice, as opposed to a concept of the core Angular API. A feature module delivers a cohesive set of functionality focused on a specific application need such as a user workflow, routing, or forms. While you can do everything within the root module, feature modules help you partition the app into focused areas. A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.

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

Assuming you already have an app that you created with the [Angular CLI](https://angular.io/cli), create a feature module using the CLI by entering the following command in the root project directory. Replace CustomerDashboard with the name of your module. You can omit the "Module" suffix from the name because the CLI appends it:

content\_copyng generate module CustomerDashboard

This causes the CLI to create a folder called customer-dashboard with a file inside called customer-dashboard.module.ts with the following contents:

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

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

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

imports: [

[CommonModule](https://angular.io/api/common/CommonModule)

],

declarations: []

})

export class CustomerDashboardModule { }

The structure of an NgModule is the same whether it is a root module or a feature module. In the CLI generated feature module, there are two JavaScript import statements at the top of the file: the first imports [NgModule](https://angular.io/api/core/NgModule), which, like the root module, lets you use the @[NgModule](https://angular.io/api/core/NgModule) decorator; the second imports [CommonModule](https://angular.io/api/common/CommonModule), which contributes many common directives such as [ngIf](https://angular.io/api/common/NgIf) and [ngFor](https://angular.io/api/common/NgForOf). Feature modules import [CommonModule](https://angular.io/api/common/CommonModule) instead of [BrowserModule](https://angular.io/api/platform-browser/BrowserModule), which is only imported once in the root module. [CommonModule](https://angular.io/api/common/CommonModule) only contains information for common directives such as [ngIf](https://angular.io/api/common/NgIf) and [ngFor](https://angular.io/api/common/NgForOf) which are needed in most templates, whereas [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) configures the Angular app for the browser which needs to be done only once.

The declarations array is available for you to add declarables, which are components, directives, and pipes that belong exclusively to this particular module. To add a component, enter the following command at the command line where customer-dashboard is the directory where the CLI generated the feature module and CustomerDashboard is the name of the component:

content\_copyng generate component customer-dashboard/CustomerDashboard

This generates a folder for the new component within the customer-dashboard folder and updates the feature module with the CustomerDashboardComponent info:

src/app/customer-dashboard/customer-dashboard.module.ts

content\_copy// import the new component

import { CustomerDashboardComponent } from './customer-dashboard/customer-dashboard.component';

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

imports: [

[CommonModule](https://angular.io/api/common/CommonModule)

],

declarations: [

CustomerDashboardComponent

],

})

The CustomerDashboardComponent is now in the JavaScript import list at the top and added to the declarations array, which lets Angular know to associate this new component with this feature module.

## **Importing a feature module**

To incorporate the feature module into your app, you have to let the root module, app.module.ts, know about it. Notice the CustomerDashboardModule export at the bottom of customer-dashboard.module.ts. This exposes it so that other modules can get to it. To import it into the AppModule, add it to the imports in app.module.ts and to the imports array:

src/app/app.module.ts

content\_copyimport { [HttpClientModule](https://angular.io/api/common/http/HttpClientModule) } from '@angular/common/[http](https://angular.io/api/common/http)';

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

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

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

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

// import the feature module here so you can add it to the imports array below

import { CustomerDashboardModule } from './customer-dashboard/customer-dashboard.module';

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

declarations: [

AppComponent

],

imports: [

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

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

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

CustomerDashboardModule // add the feature module here

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Now the AppModule knows about the feature module. If you were to add any service providers to the feature module, AppModule would know about those too, as would any other feature modules. However, NgModules don’t expose their components by default.

## **Rendering a feature module’s component template**

When the CLI generated the CustomerDashboardComponent for the feature module, it included a template, customer-dashboard.component.html, with the following markup:

src/app/customer-dashboard/customer-dashboard/customer-dashboard.component.html

content\_copy<p>

customer-dashboard works!

</p>

To see this HTML in the AppComponent, you first have to export the CustomerDashboardComponent in the CustomerDashboardModule. In customer-dashboard.module.ts, just beneath the declarations array, add an exports array containing CustomerDashboardComponent:

src/app/customer-dashboard/customer-dashboard.module.ts

content\_copyexports: [

CustomerDashboardComponent

]

Next, in the AppComponent, app.component.html, add the tag <app-customer-dashboard>:

src/app/app.component.html

content\_copy<h1>

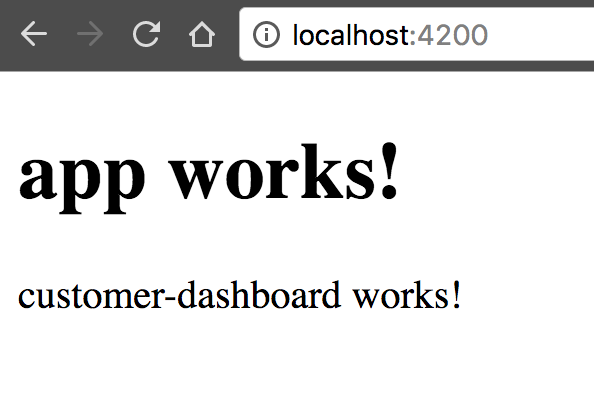
{{title}}

</h1>

<!-- add the selector from the CustomerDashboardComponent -->

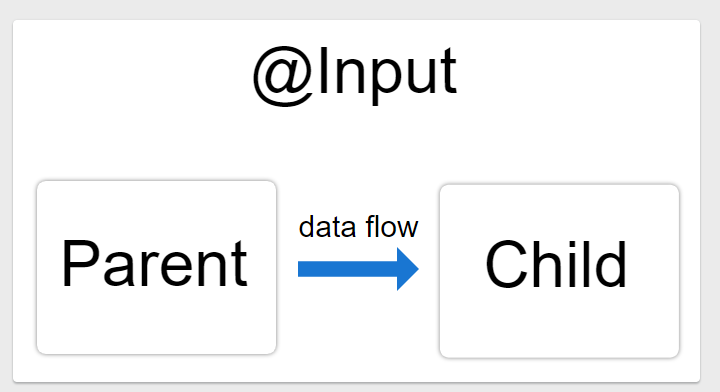
<app-customer-dashboard></app-customer-dashboard>

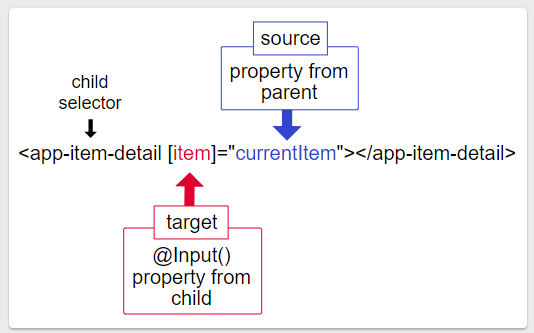
Now, in addition to the title that renders by default, the CustomerDashboardComponent template renders too:

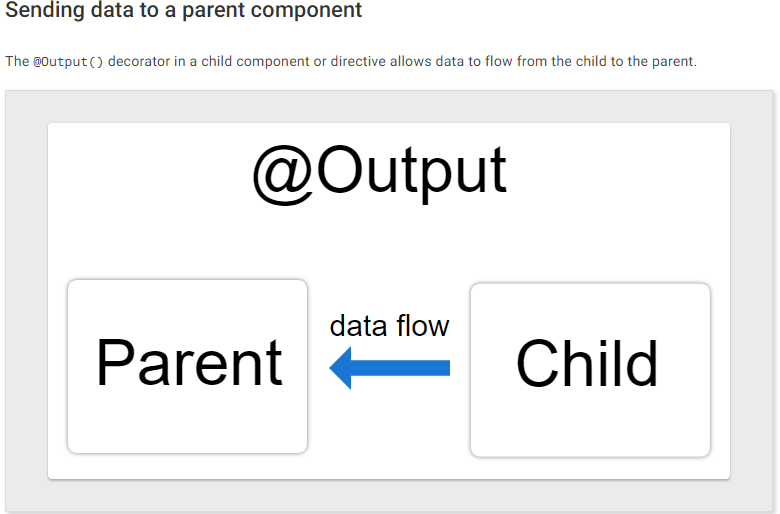


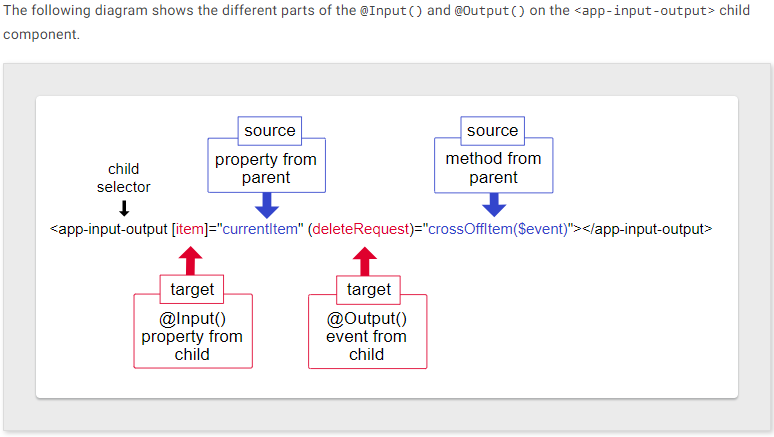
Sharing data between child and parent directives and components

<https://angular.io/guide/inputs-outputs>









<https://www.itsolutionstuff.com/post/angular-9-toastr-notifications-exampleexample.html>

Toster Alert

npm install ngx-toastr –save

npm install @angular/animations –save

**angular.json**

.....

"styles": [

"node\_modules/ngx-toastr/toastr.css",

"src/styles.css"

],

.....

In this step, we need to import ToastrModule and BrowserAnimationsModule to app.module.ts file. so let's import it as like bellow:

**src/app/app.module.ts**

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

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

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

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

**import** { ToastrModule } **from** 'ngx-toastr';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

BrowserAnimationsModule,

ToastrModule.forRoot()

],

providers: [],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }