At the core, a directive is a function that executes whenever the Angular compiler finds it in the DOM. Angular directives are used to extend the power of the HTML by giving it new syntax. Each directive has a name — either one from the Angular predefined like ng-repeat, or a custom one which can be called anything. And each directive determines where it can be used: in an element, attribute, class or comment.

 @Input() elements : {name:string, type:string} ; *//proprety*

 Lists : {name :'kareem' , type:'adel'};

     \*ngFor="let list of lists" [elements] = "list">

Understanding Shadow DOM

Before we get started and take a look at how to use Angular’s different view encapsulation types, we need to understand what Shadow DOM actually is, what it makes so awesome and why we want to use it. We won’t have a super deep dive here, since there are a lot of great resources out there already. If you want to start from scratch and learn Shadow DOM 101, which you really should in case this is new to you, [Eric Bidelman](https://twitter.com/ebidel) has written one of the best [guides](https://www.html5rocks.com/en/search?q=Shadow+DOM) over at [html5rocks.com](http://html5rocks.com/).

In one sentence, Shadow DOM is part of the Web Components standard and enables DOM tree and style encapsulation. DOM tree and style encapsulation? What does that even mean? Well, it basically means that Shadow DOM allows us to hide DOM logic behind other elements. Addition to that, it enables us to apply scoped styles to elements without them bleeding out to the outer world.

Why is that great? We can finally build components that expose a single (custom) element with hidden DOM logic under the hood, and styles that only apply to that element - a web component. Just think of an <input type="date"> element. Isn’t it nice that we can just use a single tag and the browser renders a whole date picker for us? Guess with what you can achieve that…

//directive

\*ngfor

\*ngIF

Ngclass

Ngstyle

## What are decorators?

Decorators are a design pattern that is used to separate modification or *decoration* of a class without modifying the original source code. In AngularJS, decorators are functions that allow a service, directive or filter to be modified prior to its usage.

@hostLintener:

Decorator that declares a DOM event to listen for, and provides a handler method to run when that event occurs.

# HostBinding

Decorator that marks a DOM property as a host-binding property and supplies configuration metadata. Angular automatically checks host property bindings during change detection, and if a binding changes it updates the host element of the directive.

///////////////////////////////////////////////////////////////////////////////////////

Input (@Input()) is one of the most used decorators in Angular apps. It is used to pass data from the parent or host component to the child component. This decorator has a relation with DOM property in the template where the child component is used.

# 2) Alias

By using an alias, the DOM property in the template can use different name. To give alias, we can give an argument in Input decorator such as @Input('my-alias-naming'). Below is the example

///////////////////////////////////////////////////////////////////////////////////////

Data binding is a core concept in Angular and allows to define communication between a component and the DOM, making it very easy to define interactive applications without worrying about pushing and pulling data. There are four forms of data binding and they differ in the way the data is flowing.

## From the Component to the DOM

### Interpolation: {{ value }}

### Property binding: [property]=”value”

With property binding, the value is passed from the component to the specified property, which can often be a simple html attribute:

## From the DOM to the Component

### Event binding: (event)=”function”

### Two-way data binding: [(ngModel)]=”value”

Using what’s called the banana in a box syntax, two-way data binding allows to have the data flow both ways. In this example, the user.email data property is used as the value for the input, but if the user changes the value, the component property gets updated automatically to the new value:

## ngModel Usage With Example

The ng-model directive binds the value of HTML controls (input, select, text-area) to application data. It the is a part of FormsModule. This directive is used by itself or as part of a larger form. It accepts a domain model as an optional Input. If you have a one-way binding to ngModel with [] syntax, changing the value of the domain model in the component class sets the value in the view. If you have a two-way binding with [()] syntax (also known as ‘banana-box syntax’), the value in the UI always syncs back to the domain model in your **class**.

## What is a Constructor?

A constructor is a special method which will be called whenever we create new objects. And generally used of initializing the class members. It is a feature of the class(typescript) itself, an object-oriented design concept not Angular

**What is ngOnInit() in Angular?**

1. ngOnInit is a life cycle hook managed by Angular
2. ngOnInit being added to prototype of the class created
3. ngOnInit called by Angular when component is initialized
4. Actual business logic performed ngOnInit method

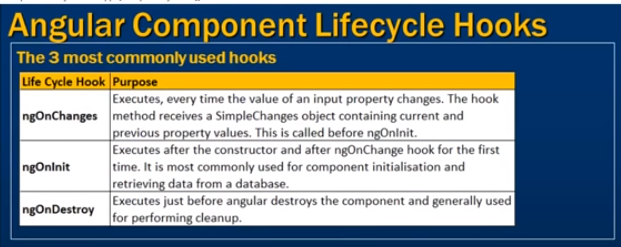
Every component we create has a life cycle managed by Angular.

One of the lifecycle hooks is ngOnInit()

**When is ngOnInit Executed?**

1. ngOnInit will be executed, When Angular done with the creating of component DOM.
2. ngOnInit will be called after the constructor execution and after first ngOnChanges

OnInit is an Angular lifecycle method, that can be hooked into components and directives in Angular



# Introduction to services and dependency injection

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

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

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

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

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

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

We might come across a situation where we need some code to be used everywhere on the page. For example, it can be for data connection that needs to be shared across components. This is achieved with the help of Services. With services, we can access methods and properties across other components in the entire project.

Dependency Injection (DI) is a core concept of Angular 2+ and allows a class receive dependencies from another class. Most of the time in Angular, dependency injection is done by injecting a service class into a component or module class.

# Dependency Injection in Angular

Dependency injection (DI), is an important application design pattern. Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity.

Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

In Angular, the DI framework provides declared dependencies to a class when that class is instantiated. This guide explains how DI works in Angular, and how you use it to make your apps flexible, efficient, and robust, as well as testable and maintainable

Table of contents

* [Angular router install](https://ultimatecourses.com/blog/angular-component-router#angular-router-install)
  + [Base href](https://ultimatecourses.com/blog/angular-component-router#base-href)
* [Router NgModule](https://ultimatecourses.com/blog/angular-component-router#router-ngmodule)
  + [RouterModule.forRoot](https://ultimatecourses.com/blog/angular-component-router#routermoduleforroot)
  + [RouterModule.forChild](https://ultimatecourses.com/blog/angular-component-router#routermoduleforchild)
* [Route Setup and Paths](https://ultimatecourses.com/blog/angular-component-router#route-setup-and-paths)
* [Router Outlet](https://ultimatecourses.com/blog/angular-component-router#router-outlet)
* [Route Params, Children, Paths](https://ultimatecourses.com/blog/angular-component-router#route-params-children-paths)
  + [Dynamic Route Params](https://ultimatecourses.com/blog/angular-component-router#dynamic-route-params)
  + [Child Routes](https://ultimatecourses.com/blog/angular-component-router#child-routes)
  + [Component-less Routes](https://ultimatecourses.com/blog/angular-component-router#component-less-routes)
  + [Using loadChildren](https://ultimatecourses.com/blog/angular-component-router#using-loadchildren)
* [Router Directives](https://ultimatecourses.com/blog/angular-component-router#router-directives)
  + [routerLink Directive](https://ultimatecourses.com/blog/angular-component-router#routerlink-directive)
  + [routerLinkActive Directive](https://ultimatecourses.com/blog/angular-component-router#routerlinkactive-directive)
* [Router Navigate](https://ultimatecourses.com/blog/angular-component-router#router-navigate)

Our root module declares our components, pipes and directives so Angular can recognize them in our templates. Our root module imports common features from the Angular 2 BrowserModule, FormsModule, and HttpModule. This allows us to use common features like ngIf and ngFor, ngModel for forms, and the Http service in our app. We're pretty set right now on basic functionality.

Another reason for encapsulating feature areas is to help keep the starting app lean and fast. When the app starts we may want to only send the parts of the app that are needed to crank it up and are the most commonly used pieces. That may not include all feature areas. The ones we want to load up front could be eagerly loaded while the others could be lazily loaded as needed.

All of these are made easier by @NgModule

#### RouterModule.forRoot

The first of the static methods is RouterModule.forRoot, which we use when defining the root config for our application in our main module.

The forRoot() function should only be used for creating the root modules' routes. When we get to child modules and routes, we'll use other techniques that we'll see in later posts*.*

We use **path: ''** to match the empty path, i.e. **https://yourdomain.com**

### Router Outlet

After configuring our routes, the next step is to tell Angular where to load the components using a directive named router-outlet. When the router has matched a route and found the component(s) to load, it will dynamically create our component and inject it as a sibling alongside the router-outlet element.

#### Dynamic Route Params:to make the root dynamic instated of static For example, if we wanted to have user “profile” pages with their unique “username” in the route, we could define the path like this:

A component in Angular is a TypeScript class decorated with the @Component decorator. It has an attached template and CSS stylesheets that form the component’s view.

The Angular router is a core part of the Angular platform. It enables developers to build Single Page Applications with multiple views and allow navigation between these views.

#### THE ROUTER-OUTLET

The [Router-Outlet](https://angular.io/api/router/RouterOutlet) is a directive that’s available from the router library where the Router inserts the component that gets matched based on the current browser’s URL. You can add multiple outlets in your Angular application which enables you to implement advanced routing scenari

The path can take a **wildcard** string (\*\*). The router will select this route if the requested URL doesn’t match any paths for the defined routes. This can be used for displaying a “Not Found” view or redirecting to a specific view if no match is found.

#### ROUTE PARAMS

Creating routes with parameters is a common feature in web apps. Angular Router allows you to access parameters in different ways:

* Using the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) service,
* Using the [ParamMap](https://angular.io/api/router/ParamMap) observable available starting with v4.

#### ROUTE GUARDS

A route guard is a feature of the Angular Router that allows developers to run some logic when a route is requested, and based on that logic, it allows or denies the user access to the route. It’s commonly used to check if a user is logged in and has the authorization before he can access a page.

Angular Router supports multiple outlets in the same application.

A component has one associated primary route and can have auxiliary routes. Auxiliary routes enable developers to navigate multiple routes at the same time.

To create an auxiliary route, you’ll need a named router outlet where the component associated with the auxiliary route will be displayed.

* The outlet with no name is the primary outlet.
* All outlets should have a name except for the primary outlet.

The [RouterOutlet](https://angular.io/api/router/RouterOutlet) is a directive from the router library that is used like a component. It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet.

* Organizing the application features into modules.
* Navigating to a component (*Heroes* link to "Heroes List").
* Including a route parameter (passing the Hero id while routing to the "Hero Detail").
* Child routes (the *Crisis Center* has its own routes).
* The [CanActivate](https://angular.io/api/router/CanActivate) guard (checking route access).
* The [CanActivateChild](https://angular.io/api/router/CanActivateChild) guard (checking child route access).
* The [CanDeactivate](https://angular.io/api/router/CanDeactivate) guard (ask permission to discard unsaved changes).
* The [Resolve](https://angular.io/api/router/Resolve) guard (pre-fetching route data).
* Lazy loading feature modules.
* The [CanLoad](https://angular.io/api/router/CanLoad) guard (check before loading feature module assets).

# ParamMap[link](https://angular.io/api/router/ParamMap#parammap)

A map that provides access to the required and optional parameters specific to a route. The map supports retrieving a single value with get() or multiple values with getAll().

paramMap Observable

paramMap is the method of ActivatedRoute and it will return an observable and it will give data only if we subscribe to it. The argument in the subscribe method is an arrow function. The observable provides the parameter which we strongly typed to paramMap. paramMap is coming from Router package. We should have imported the Router package.

**How to Get Route Parameters**

The Angular Router provides two different methods to get route parameters:

* Using the route snapshot,
* Using Router Observables

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

### Angular Auth Guards.

Angular **Auth Guards** add authentication layer which helps in preventing loading of components in various scenarios like Login authentication, permission-based authentication etc. We can add checks before loading components using Angular guards which are of four types

**CanActivate:** Checks route navigation before the component is loaded.

**CanActivateChild**: Checks route children navigation before the component is loaded.

**CanDeactivate**: Checks navigation from the current route eg leaving partially filled form.

**Resolve**: Resolve loads/ retrieves data before the route is activated.

**CanLoad**: Checks to see if a user can route to a module that lazy loaded.