***Angular Question Bank:***

1. **What is the difference between AngularJS and Angular?**
2. [**What are the key components of Angular?**](https://github.com/sudheerj/angular-interview-questions#what-are-the-key-components-of-angular)
3. **What are directives?**
4. **What are components?**
5. **What are lifecycle hooks available?**
6. **What are the differences between Component and Directive?**
7. **What is a template?**
8. **What is a module?**
9. **What is a data binding?**
10. **What is metadata?**
11. **What is angular CLI?**
12. **What is the difference between constructor and ngOnInit?**
13. **What is a service?**
14. **What is dependency injection in Angular?**
15. **What is the purpose of async pipe?**
16. **What is the option to choose between inline and external template file?**
17. **What is the purpose of ngIf directive?**
18. **What is interpolation?**
19. **What are template expressions?**
20. **What are template statements?**
21. **What are pipes?**
22. **What is a parameterized pipe?**
23. **What is a AppModule?**
24. **What are observables?**
25. **What is HttpClient and its benefits?**
26. **What is subscribing?**
27. **How do you perform Error handling?**
28. **What is an observer?**
29. **What is the difference between promise and observable?**
30. **How do you perform error handling in observables?**
31. **What are the utility functions provided by RxJS?**
32. **What are angular elements?**
33. **What are custom elements?**
34. **Explain how custom elements works internally?**
35. **What are the mapping rules between Angular component and custom element?**
36. **What are dynamic components?**
37. **What are the various kinds of directives?**
38. **How do you create directives using CLI?**
39. **Give an example for attribute directives?**
40. **What is Angular Router?**
41. **What is the purpose of base href tag?**
42. **What are the router imports?**
43. **What are router links?**
44. **What are active router links?**
45. **What are router events?**
46. **What is activated route?**
47. **How do you define routes?**
48. **Do we need a Routing Module always?**
49. **What is Angular Universal?**
50. **What are different types of compilation in Angular?**
51. **What is JIT ?**
52. **What is AOT ?**
53. **Why do we need compilation process?**
54. **What are the three phases of AOT?**
55. **What happens if we import the same module twice?**
56. **What is lazy loading?**
57. **How does angular finds components, directives and pipes?**
58. **What are reactive forms?**
59. **What are template driven forms?**
60. **What are the differences between reactive forms and template driven forms?**
61. **What are the state CSS classes provided by ngModel?**
62. **How to set ngFor and ngIf on the same element?**

***The Questions along with answers:***

1. **What is the difference between AngularJS and Angular?**

Angular is a completely revived component-based framework in which an application is a tree of individual components.

Some of the major difference in tabular form

| **AngularJS** | **Angular** |
| --- | --- |
| It is based on MVC architecture | This is based on Service/Controller |
| It uses JavaScript to build the application | Introduced the TypeScript to write the application |
| Based on controllers concept | This is a component based UI approach |
| Not a mobile friendly framework | Developed considering mobile platform |
| Difficulty in SEO friendly application development | Ease to create SEO friendly applications |

### What are the key components of Angular?

Angular has the below key components,

* 1. **Component:** These are the basic building blocks of angular application to control HTML views.
  2. **Modules:** An angular module is set of angular basic building blocks like component, directives, services etc. An application is divided into logical pieces and each piece of code is called as "module" which perform a single task.
  3. **Templates:** This represent the views of an Angular application.
  4. **Services:** It is used to create components which can be shared across the entire application.
  5. **Metadata:** This can be used to add more data to an Angular class.

### What are directives?

Directives add behaviour to an existing DOM element or an existing component instance.

import { Directive, ElementRef, Input } from '@angular/core';

@Directive({ selector: '[myHighlight]' })

export class HighlightDirective {

constructor(el: ElementRef) {

el.nativeElement.style.backgroundColor = 'yellow';

}

}

Now this directive extends HTML element behavior with a yellow background as below

<p myHighlight>Highlight me!</p>

1. **What are components?**

Components are the most basic UI building block of an Angular app which formed a tree of Angular components. These components are subset of directives. Unlike directives, components always have a template and only one component can be instantiated per an element in a template. Let's see a simple example of Angular component

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

@Component ({

selector: 'my-app',

template: ` <div>

<h1>{{title}}</h1>

<div>Learn Angular6 with examples</div>

</div> `,

})

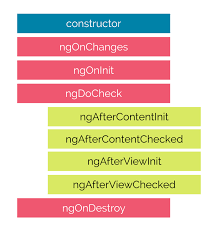
export class AppComponent {

title: string = 'Welcome to Angular world';

}

1. **What are lifecycle hooks available?**

Angular application goes through an entire set of processes or has a lifecycle right from its initiation to the end of the application. The representation of lifecycle in pictorial representation as follows,

[](https://github.com/sudheerj/angular-interview-questions/blob/master/images/lifecycle.png)

The description of each lifecycle method is as below,

* 1. **ngOnChanges:** When the value of a data bound property changes, then this method is called.
  2. **ngOnInit:** This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
  3. **ngDoCheck:** This is for the detection and to act on changes that Angular can't or won't detect on its own.
  4. **ngAfterContentInit:** This is called in response after Angular projects external content into the component's view.
  5. **ngAfterContentChecked:** This is called in response after Angular checks the content projected into the component.
  6. **ngAfterViewInit:** This is called in response after Angular initializes the component's views and child views.
  7. **ngAfterViewChecked:** This is called in response after Angular checks the component's views and child views.
  8. **ngOnDestroy:** This is the cleanup phase just before Angular destroys the directive/component.

### What are the differences between Component and Directive?

In a short note, A component(@component) is a directive-with-a-template.

Some of the major differences are mentioned in a tabular form

| **Component** | **Directive** |
| --- | --- |
| To register a component we use @Component meta-data annotation | To register directives we use @Directive meta-data annotation |
| Components are typically used to create UI widgets | Directive is used to add behavior to an existing DOM element |
| Component is used to break up the application into smaller components | Directive is use to design re-usable components |
| Only one component can be present per DOM element | Many directives can be used per DOM element |
| @View decorator or templateurl/template are mandatory | Directive doesn't use View |

### What is a template?

A template is a HTML view where you can display data by binding controls to properties of an Angular component. You can store your component's template in one of two places. You can define it inline using the template property, or you can define the template in a separate HTML file and link to it in the component metadata using the @Component decorator's templateUrl property.

**Using inline template with template syntax,**

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

@Component ({

selector: 'my-app',

template: '

<div>

<h1>{{title}}</h1>

<div>Learn Angular</div>

</div>

'

})

export class AppComponent {

title: string = 'Hello World';

}

**Using separate template file such as app.component.html**

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

@Component ({

selector: 'my-app',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

title: string = 'Hello World';

}

1. **What is a module?**

Modules are logical boundaries in your application and the application is divided into separate modules to separate the functionality of your application. Lets take an example of **app.module.ts** root module declared with **@NgModule** decorator as below,

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

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

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

@NgModule ({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ],

providers: []

})

export class AppModule { }

The NgModule decorator has five important(among all) options

* 1. The imports option is used to import other dependent modules. The BrowserModule is required by default for any web based angular application
  2. The declarations option is used to define components in the respective module
  3. The bootstrap option tells Angular which Component to bootstrap in the application
  4. The providers option is used to configure set of injectable objects that are available in the injector of this module.
  5. The entryComponents option is a set of components dynamically loaded into the view.

1. **What is a data binding?**

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(divided as 3 categories) which differ in the way the data is flowing.

* 1. **From the Component to the DOM:**

**Interpolation:** {{ value }}: Adds the value of a property from the component

<li>Name: {{ user.name }}</li>

<li>Address: {{ user.address }}</li>

**Property binding:** [property]=”value”: The value is passed from the component to the specified property or simple HTML attribute

<input type="email" [value]="user.email">

* 1. **From the DOM to the Component:** **Event binding: (event)=”function”:** When a specific DOM event happens (eg.: click, change, keyup), call the specified method in the component

<button (click)="logout()"></button>

* 1. **Two-way binding:** **Two-way data binding:** [(ngModel)]=”value”: Two-way data binding allows to have the data flow both ways. For example, in the below code snippet, both the email DOM input and component email property are in sync

<input type="email" [(ngModel)]="user.email">

### What is metadata?

Metadata is used to decorate a class so that it can configure the expected behavior of the class. The metadata is represented by decorators

* 1. **Class decorators**, e.g. @Component and @NgModule
  2. import { NgModule, Component } from '@angular/core';
  3. @Component({
  4. selector: 'my-component',
  5. template: '<div>Class decorator</div>',
  6. })
  7. export class MyComponent {
  8. constructor() {
  9. console.log('Hey I am a component!');
  10. }
  11. }
  12. @NgModule({
  13. imports: [],
  14. declarations: [],
  15. })
  16. export class MyModule {
  17. constructor() {
  18. console.log('Hey I am a module!');
  19. }

}

* 1. **Property decorators** Used for properties inside classes, e.g. @Input and @Output
  2. import { Component, Input } from '@angular/core';
  3. @Component({
  4. selector: 'my-component',
  5. template: '<div>Property decorator</div>'
  6. })
  7. export class MyComponent {
  8. @Input()
  9. title: string;

}

* 1. **Method decorators** Used for methods inside classes, e.g. @HostListener
  2. import { Component, HostListener } from '@angular/core';
  3. @Component({
  4. selector: 'my-component',
  5. template: '<div>Method decorator</div>'
  6. })
  7. export class MyComponent {
  8. @HostListener('click', ['$event'])
  9. onHostClick(event: Event) {
  10. // clicked, `event` available
  11. }

}

* 1. **Parameter decorators** Used for parameters inside class constructors, e.g. @Inject, Optional
  2. import { Component, Inject } from '@angular/core';
  3. import { MyService } from './my-service';
  4. @Component({
  5. selector: 'my-component',
  6. template: '<div>Parameter decorator</div>'
  7. })
  8. export class MyComponent {
  9. constructor(@Inject(MyService) myService) {
  10. console.log(myService); // MyService
  11. }

}

### What is angular CLI?

Angular CLI(**Command Line Interface**) is a command line interface to scaffold and build angular apps using nodejs style (commonJs) modules. You need to install using below npm command,

npm install @angular/cli@latest

Below are the list of few commands, which will come handy while creating angular projects

* 1. **Creating New Project:** ng new
  2. **Generating Components, Directives & Services:** ng generate/g The different types of commands would be,
     1. ng generate class my-new-class: add a class to your application
     2. ng generate component my-new-component: add a component to your application
     3. ng generate directive my-new-directive: add a directive to your application
     4. ng generate enum my-new-enum: add an enum to your application
     5. ng generate module my-new-module: add a module to your application
     6. ng generate pipe my-new-pipe: add a pipe to your application
     7. ng generate service my-new-service: add a service to your application
  3. **Running the Project:** ng serve

### What is the difference between constructor and ngOnInit?

TypeScript classes has a default method called constructor which is normally used for the initialization purpose. Whereas ngOnInit method is specific to Angular, especially used to define Angular bindings. Even though constructor getting called first, it is preferred to move all of your Angular bindings to ngOnInit method. In order to use ngOnInit, you need to implement OnInit interface as below,

export class App implements OnInit{

constructor(){

//called first time before the ngOnInit()

}

ngOnInit(){

//called after the constructor and called after the first ngOnChanges()

}

}

### What is a service?

A service is used when a common functionality needs to be provided to various modules. Services allow for greater separation of concerns for your application and better modularity by allowing you to extract common functionality out of components.

Let's create a repoService which can be used across components,

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

import { Http } from '@angular/http';

@Injectable({ // The Injectable decorator is required for dependency injection to work

// providedIn option registers the service with a specific NgModule

providedIn: 'root', // This declares the service with the root app (AppModule)

})

export class RepoService{

constructor(private http: Http){

}

fetchAll(){

return this.http.get('https://api.github.com/repositories');

}

}

The above service uses Http service as a dependency.

### What is dependency injection in Angular?

Dependency injection (DI), is an important application design pattern in which a class asks for dependencies from external sources rather than creating them itself. Angular comes with its own dependency injection framework for resolving dependencies( services or objects that a class needs to perform its function).So you can have your services depend on other services throughout your application.

### What is the purpose of async pipe?

The AsyncPipe subscribes to an observable or promise and returns the latest value it has emitted. When a new value is emitted, the pipe marks the component to be checked for changes.

Let's take a time observable which continuously updates the view for every 2 seconds with the current time.

@Component({

selector: 'async-observable-pipe',

template: `<div><code>observable|async</code>:

Time: {{ time | async }}</div>`

})

export class AsyncObservablePipeComponent {

time = new Observable(observer =>

setInterval(() => observer.next(new Date().toString()), 2000)

);

}

### What is the option to choose between inline and external template file?

You can store your component's template in one of two places. You can define it inline using the **template** property, or you can define the template in a separate HTML file and link to it in the component metadata using the **@Component** decorator's **templateUrl** property.

The choice between inline and separate HTML is a matter of taste, circumstances, and organization policy. But normally we use inline template for small portion of code and external template file for bigger views. By default, the Angular CLI generates components with a template file. But you can override that with the below command,

ng generate component hero -it

1. **What is the purpose of ngIf directive?**

Sometimes an app needs to display a view or a portion of a view only under specific circumstances. The Angular ngIf directive inserts or removes an element based on a truthy/falsy condition. Let's take an example to display a message if the user age is more than 18,

<p \*ngIf="user.age > 18">You are not eligible for student pass!</p>

**Note:** Angular isn't showing and hiding the message. It is adding and removing the paragraph element from the DOM. That improves performance, especially in the larger projects with many data bindings.

### What is interpolation?

Interpolation is a special syntax that Angular converts into property binding. It’s a convenient alternative to property binding. It is represented by double curly braces({{}}). The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property.

Let's take an example,

<h3>

{{title}}

<img src="{{url}}" style="height:30px">

</h3>

In the example above, Angular evaluates the title and url properties and fills in the blanks, first displaying a bold application title and then a URL.

1. **What are template expressions?**

A template expression produces a value similar to any Javascript expression. Angular executes the expression and assigns it to a property of a binding target; the target might be an HTML element, a component, or a directive. In the property binding, a template expression appears in quotes to the right of the = symbol as in [property]="expression". In interpolation syntax, the template expression is surrounded by double curly braces. For example, in the below interpolation, the template expression is {{username}},

<h3>{{username}}, welcome to Angular</h3>

The below javascript expressions are prohibited in template expression

1. assignments (=, +=, -=, ...)
2. new
3. chaining expressions with ; or ,
4. increment and decrement operators (++ and --)
5. **What are template statements?**

A template statement responds to an event raised by a binding target such as an element, component, or directive. The template statements appear in quotes to the right of the = symbol like **(event)="statement"**.

Let's take an example of button click event's statement

<button (click)="editProfile()">Edit Profile</button>

In the above expression, editProfile is a template statement. The below JavaScript syntax expressions are not allowed.

1. new
2. increment and decrement operators, ++ and --
3. operator assignment, such as += and -=
4. the bitwise operators | and &
5. the template expression operators

### What are pipes?

A pipe takes in data as input and transforms it to a desired output. For example, let us take a pipe to transform a component's birthday property into a human-friendly date using **date** pipe.

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

@Component({

selector: 'app-birthday',

template: `<p>Birthday is {{ birthday | date }}</p>`

})

export class BirthdayComponent {

birthday = new Date(1987, 6, 18); // June 18, 1987

}

### What is a parameterized pipe?

A pipe can accept any number of optional parameters to fine-tune its output. The parameterized pipe can be created by declaring the pipe name with a colon ( : ) and then the parameter value. If the pipe accepts multiple parameters, separate the values with colons. Let's take a birthday example with a particular format(dd/MM/yyyy):

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

@Component({

selector: 'app-birthday',

template: `<p>Birthday is {{ birthday | date:'dd/MM/yyyy'}}</p>` // 18/06/1987

})

export class BirthdayComponent {

birthday = new Date(1987, 6, 18);

}

**Note:** The parameter value can be any valid template expression, such as a string literal or a component property.

### What is a AppModule?

Every application has at least one Angular module, the root module that you bootstrap to launch the application is called as bootstrapping module. It is commonly known as AppModule. The default structure of AppModule generated by AngularCLI would be as follows,

```javascript

/\* JavaScript imports \*/

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

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

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

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

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

/\* the AppModule class with the @NgModule decorator \*/

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

```

1. **What are observables?**

Observables are declarative which provide support for passing messages between publishers and subscribers in your application. They are mainly used for event handling, asynchronous programming, and handling multiple values. In this case, you define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe.

1. **What is HttpClient and its benefits?**

Most of the Front-end applications communicate with backend services over HTTP protocol using either XMLHttpRequest interface or the fetch() API. Angular provides a simplified client HTTP API known as **HttpClient** which is based on top of XMLHttpRequest interface. This client is avaialble from @angular/common/http package. You can import in your root module as below,

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

The major advantages of HttpClient can be listed as below,

* 1. Contains testability features
  2. Provides typed request and response objects
  3. Intercept request and response
  4. Supports Observalbe APIs
  5. Supports streamlined error handling

### What is subscribing?

An Observable instance begins publishing values only when someone subscribes to it. So you need to subscribe by calling the **subscribe()** method of the instance, passing an observer object to receive the notifications.

Let's take an example of creating and subscribing to a simple observable, with an observer that logs the received message to the console.

Creates an observable sequence of 5 integers, starting from 1

const source = range(1, 5);

// Create observer object

const myObserver = {

next: x => console.log('Observer got a next value: ' + x),

error: err => console.error('Observer got an error: ' + err),

complete: () => console.log('Observer got a complete notification'),

};

// Execute with the observer object and Prints out each item

source.subscribe(myObserver);

// => Observer got a next value: 1

// => Observer got a next value: 2

// => Observer got a next value: 3

// => Observer got a next value: 4

// => Observer got a next value: 5

// => Observer got a complete notification

### How do you perform Error handling?

If the request fails on the server or failed to reach the server due to network issues then HttpClient will return an error object instead of a successful reponse. In this case, you need to handle in the component by passing error object as a second callback to subscribe() method.

Let's see how it can be handled in the component with an example,

fetchUser() {

this.userService.getProfile()

.subscribe(

(data: User) => this.userProfile = { ...data }, // success path

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

);

}

It is always a good idea to give the user some meaningful feedback instead of displaying the raw error object returned from HttpClient.

### What is an observer?

Observer is an interface for a consumer of push-based notifications delivered by an Observable. It has below structure,

interface Observer<T> {

closed?: boolean;

next: (value: T) => void;

error: (err: any) => void;

complete: () => void;

}

A handler that implements the Observer interface for receiving observable notifications will be passed as a parameter for observable as below,

myObservable.subscribe(myObserver);

**Note:** If you don't supply a handler for a notification type, the observer ignores notifications of that type.

1. **What is the difference between promise and observable?**

Below are the list of differences between promise and observable,

| **Observable** | **Promise** |
| --- | --- |
| Declarative: Computation does not start until subscription so that they can be run whenever you need the result | Execute immediately on creation |
| Provide multiple values over time | Provide only one |
| Subscribe method is used for error handling which makes centralized and predictable error handling | Push errors to the child promises |
| Provides chaining and subscription to handle complex applications | Uses only .then() clause |

### How do you perform error handling in observables?

You can handle errors by specifying an **error callback** on the observer instead of relying on try/catch which are ineffective in asynchronous environment.

For example, you can define error callback as below,

myObservable.subscribe({

next(num) { console.log('Next num: ' + num)},

error(err) { console.log('Received an errror: ' + err)}

});

### What are the utility functions provided by RxJS?

The RxJS library also provides below utility functions for creating and working with observables.

* 1. Converting existing code for async operations into observables
  2. Iterating through the values in a stream
  3. Mapping values to different types
  4. Filtering streams
  5. Composing multiple streams

### What are angular elements?

Angular elements are Angular components packaged as **custom elements**(a web standard for defining new HTML elements in a framework-agnostic way). Angular Elements hosts an Angular component, providing a bridge between the data and logic defined in the component and standard DOM APIs, thus, providing a way to use Angular components in non-Angular environments.

### What are custom elements?

Custom elements (or Web Components) are a Web Platform feature which extends HTML by allowing you to define a tag whose content is created and controlled by JavaScript code. The browser maintains a CustomElementRegistry of defined custom elements, which maps an instantiable JavaScript class to an HTML tag. Currently this feature is supported by Chrome, Firefox, Opera, and Safari, and available in other browsers through polyfills.

### Explain how custom elements works internally?

Below are the steps in an order about custom elements functionality,

1. **App registers custom element with browser:** Use the createCustomElement() function to convert a component into a class that can be registered with the browser as a custom element.
2. **App adds custom element to DOM:** Add custom element just like a built-in HTML element directly into the DOM.
3. **Browser instantiate component based class:** Browser creates an instance of the registered class and adds it to the DOM.
4. **Instance provides content with data binding and change detection:** The content with in template is rendered using the component and DOM data. The flow chart of the custom elements functionality would be as follows,

### What are the mapping rules between Angular component and custom element?

The Component properties and logic maps directly into HTML attributes and the browser's event system. Let us describe them in two steps,

* 1. The createCustomElement() API parses the component input properties with corresponding attributes for the custom element. For example, component @Input('myInputProp') converted as custom element attribute my-input-prop.
  2. The Component outputs are dispatched as HTML Custom Events, with the name of the custom event matching the output name. For example, component @Output() valueChanged = new EventEmitter() converted as custom element with dispatch event as "valueChanged".

[](https://github.com/sudheerj/angular-interview-questions/blob/master/images/customElement.png)

### What are dynamic components?

Dynamic components are the components in which components location in the application is not defined at build time.i.e, They are not used in any angular template. But the component is instantiated and placed in the application at runtime.

### What are the various kinds of directives?

There are mainly three kinds of directives,

* 1. **Components** — These are directives with a template.
  2. **Structural directives** — These directives change the DOM layout by adding and removing DOM elements.
  3. **Attribute directives** — These directives change the appearance or behavior of an element, component, or another directive.

### How do you create directives using CLI?

You can use CLI command ng generate directive to create the directive class file. It creates the source file(src/app/components/directivename.directive.ts), the respective test file(.spec.ts) and declare the directive class file in root module.

### Give an example for attribute directives?

Let's take simple highlighter behavior as a example directive for DOM element. You can create and apply the attribute directive using below steps,

* 1. Create HighlightDirective class with the file name src/app/highlight.directive.ts. In this file, we need to import **Directive** from core library to apply the metadata and **ElementRef** in the directive's constructor to inject a reference to the host DOM element ,
  2. import { Directive, ElementRef } from '@angular/core';
  3. @Directive({
  4. selector: '[appHighlight]'
  5. })
  6. export class HighlightDirective {
  7. constructor(el: ElementRef) {
  8. el.nativeElement.style.backgroundColor = 'red';
  9. }

}

* 1. Apply the attribute directive as an attribute to the host element(for example,

)

<p appHighlight>Highlight me!</p>

* 1. Run the application to see the highlight behavior on paragraph element

ng serve

### What is Angular Router?

Angular Router is a mechanism in which navigation happens from one view to the next as users perform application tasks. It borrows the concepts or model of browser's application navigation.

### What is the purpose of base href tag?

The routing application should add element to the index.html as the first child in the tag in order to indicate how to compose navigation URLs. If app folder is the application root then you can set the href value as below

<base href="/">

### What are the router imports?

The Angular Router which represents a particular component view for a given URL is not part of Angular Core. It is available in library named @angular/router to import required router components. For example, we import them in app module as below,

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

### What are router links?

The RouterLink is a directive on the anchor tags give the router control over those elements. Since the navigation paths are fixed, you can assign string values to router-link directive as below,

<h1>Angular Router</h1>

<nav>

<a routerLink="/todosList" >List of todos</a>

<a routerLink="/completed" >Completed todos</a>

</nav>

<router-outlet></router-outlet>

### What are active router links?

RouterLinkActive is a directive that toggles css classes for active RouterLink bindings based on the current RouterState. i.e, the Router will add CSS classes when this link is active and and remove when the link is inactive. For example, you can add them to RouterLinks as below

<h1>Angular Router</h1>

<nav>

<a routerLink="/todosList" routerLinkActive="active">List of todos</a>

<a routerLink="/completed" routerLinkActive="active">Completed todos</a>

</nav>

<router-outlet></router-outlet>

### What are router events?

During each navigation, the Router emits navigation events through the Router.events property allowing you to track the lifecycle of the route.

The sequence of router events is as below,

* 1. NavigationStart,
  2. RouteConfigLoadStart,
  3. RouteConfigLoadEnd,
  4. RoutesRecognized,
  5. GuardsCheckStart,
  6. ChildActivationStart,
  7. ActivationStart,
  8. GuardsCheckEnd,
  9. ResolveStart,
  10. ResolveEnd,
  11. ActivationEnd
  12. ChildActivationEnd
  13. NavigationEnd,
  14. NavigationCancel,
  15. NavigationError
  16. Scroll

### What is activated route?

ActivatedRoute contains the information about a route associated with a component loaded in an outlet. It can also be used to traverse the router state tree. The ActivatedRoute will be injected as a router service to access the information. In the below example, you can access route path and parameters,

@Component({...})

class MyComponent {

constructor(route: ActivatedRoute) {

const id: Observable<string> = route.params.pipe(map(p => p.id));

const url: Observable<string> = route.url.pipe(map(segments => segments.join('')));

// route.data includes both `data` and `resolve`

const user = route.data.pipe(map(d => d.user));

}

}

### How do you define routes?

A router must be configured with a list of route definitions. You configures the router with routes via the RouterModule.forRoot() method, and adds the result to the AppModule's imports array.

const appRoutes: Routes = [

{ path: 'todo/:id', component: TodoDetailComponent },

{

path: 'todos',

component: TodosListComponent,

data: { title: 'Todos List' }

},

{ path: '',

redirectTo: '/todos',

pathMatch: 'full'

},

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

];

@NgModule({

imports: [

RouterModule.forRoot(

appRoutes,

{ enableTracing: true } // <-- debugging purposes only

)

// other imports here

],

...

})

export class AppModule { }

### Do we need a Routing Module always?

No, the Routing Module is a design choice. You can skip routing Module (for example, AppRoutingModule) when the configuration is simple and merge the routing configuration directly into the companion module (for example, AppModule). But it is recommended when the configuration is complex and includes specialized guard and resolver services.

### What is Angular Universal?

Angular Universal is a server-side rendering module for Angular applications in various scenarios. This is a community driven project and available under @angular/platform-server package. Recently Angular Universal is integrated with Angular CLI.

### What are different types of compilation in Angular?

Angular offers two ways to compile your application,

* 1. Just-in-Time (JIT)
  2. Ahead-of-Time (AOT)

### What is JIT?

Just-in-Time (JIT) is a type of compilation that compiles your app in the browser at runtime. JIT compilation is the default when you run the ng build (build only) or ng serve (build and serve locally) CLI commands. i.e, the below commands used for JIT compilation,

ng build

ng serve

### What is AOT?

Ahead-of-Time (AOT) is a type of compilation that compiles your app at build time. For AOT compilation, include the --aot option with the ng build or ng serve command as below,

ng build --aot

ng serve --aot

**Note:** The ng build command with the --prod meta-flag (ng build --prod) compiles with AOT by default.

### Why do we need compilation process?

The Angular components and templates cannot be understood by the browser directly. Due to that Angular applications require a compilation process before they can run in a browser. For example, In AOT compilation, both Angular HTML and TypeScript code converted into efficient JavaScript code during the build phase before browser runs it.

### What are the three phases of AOT?

The AOT compiler works in three phases,

* 1. **Code Analysis:** The compiler records a representation of the source
  2. **Code generation:** It handles the interpretation as well as places restrictions on what it interprets.
  3. **Validation:** In this phase, the Angular template compiler uses the TypeScript compiler to validate the binding expressions in templates.

### What happens if we import the same module twice?

If multiple modules imports the same module then angular evaluates it only once (When it encounters the module first time). It follows this condition even the module appears at any level in a hierarchy of imported NgModules.

### What is lazy loading?

Lazy loading is one of the most useful concepts of Angular Routing. It helps us to download the web pages in chunks instead of downloading everything in a big bundle. It is used for lazy loading by asynchronously loading the feature module for routing whenever required using the property loadChildren. Let's load both Customer and Order feature modules lazily as below,

const routes: Routes = [

{

path: 'customers',

loadChildren: () => import('./customers/customers.module').then(module => module.CustomersModule)

},

{

path: 'orders',

loadChildren: () => import('./orders/orders.module').then(module => module.OrdersModule)

},

{

path: '',

redirectTo: '',

pathMatch: 'full'

}

];

### How does angular finds components, directives and pipes?

The Angular compiler finds a component or directive in a template when it can match the selector of that component or directive in that template. Whereas it finds a pipe if the pipe's name appears within the pipe syntax of the template HTML.

### What are reactive forms?

Reactive forms is a model-driven approach for creating forms in a reactive style(form inputs changes over time). These are built around observable streams, where form inputs and values are provided as streams of input values. Let's follow the below steps to create reactive forms,

1. Register the reactive forms module which declares reactive-form directives in your app
2. import { ReactiveFormsModule } from '@angular/forms';
3. @NgModule({
4. imports: [
5. // other imports ...
6. ReactiveFormsModule
7. ],
8. })

export class AppModule { }

1. Create a new FormControl instance and save it in the component.
2. import { Component } from '@angular/core';
3. import { FormControl } from '@angular/forms';
4. @Component({
5. selector: 'user-profile',
6. styleUrls: ['./user-profile.component.css']
7. })
8. export class UserProfileComponent {
9. userName = new FormControl('');

}

1. Register the FormControl in the template.
2. <label>
3. User name:
4. <input type="text" [formControl]="userName">

</label>

Finally, the component with reactive form control appears as below, ```js import { Component } from '@angular/core'; import { FormControl } from '@angular/forms';

@Component({

selector: 'user-profile',

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

template: `

<label>

User name:

<input type="text" [formControl]="userName">

</label>

`

})

export class UserProfileComponent {

userName = new FormControl('');

}

```

### What are template driven forms?

Template driven forms are model-driven forms where you write the logic, validations, controls etc, in the template part of the code using directives. They are suitable for simple scenarios and uses two-way binding with [(ngModel)] syntax. For example, you can create register form easily by following the below simple steps,

1. Import the FormsModule into the Application module's imports array
2. import { BrowserModule } from '@angular/platform-browser';
3. import { NgModule } from '@angular/core';
4. import {FormsModule} from '@angular/forms'
5. import { RegisterComponent } from './app.component';
6. @NgModule({
7. declarations: [
8. RegisterComponent,
9. ],
10. imports: [
11. BrowserModule,
12. FormsModule
13. ],
14. providers: [],
15. bootstrap: [RegisterComponent]
16. })

export class AppModule { }

1. Bind the form from template to the component using ngModel syntax
2. <input type="text" class="form-control" id="name"
3. required

[(ngModel)]="model.name" name="name">

1. Attach NgForm directive to the tag in order to create FormControl instances and register them

<form #registerForm="ngForm">

1. Apply the validation message for form controls
2. <label for="name">Name</label>
3. <input type="text" class="form-control" id="name"
4. required
5. [(ngModel)]="model.name" name="name"
6. #name="ngModel">
7. <div [hidden]="name.valid || name.pristine"
8. class="alert alert-danger">
9. Please enter your name

</div>

1. Let's submit the form with ngSubmit directive and add type="submit" button at the bottom of the form to trigger form submit.
2. <form (ngSubmit)="onSubmit()" #heroForm="ngForm">
3. // Form goes here

<button type="submit" class="btn btn-success" [disabled]="!registerForm.form.valid">Submit</button>

Finally, the completed template-driven registration form will be appeared as follow.

```html

<div class="container">

<h1>Registration Form</h1>

<form (ngSubmit)="onSubmit()" #registerForm="ngForm">

<div class="form-group">

<label for="name">Name</label>

<input type="text" class="form-control" id="name"

required

[(ngModel)]="model.name" name="name"

#name="ngModel">

<div [hidden]="name.valid || name.pristine"

class="alert alert-danger">

Please enter your name

</div>

</div>

<button type="submit" class="btn btn-success" [disabled]="!registerForm.form.valid">Submit</button>

</form>

</div>

1. **What are the differences between reactive forms and template driven forms?**

Below are the main differences between reactive forms and template driven forms

| **Feature** | **Reactive** | **Template-Driven** |
| --- | --- | --- |
| Form model setup | Created(FormControl instance) in component explicitly | Created by directives |
| Data updates | Synchronous | Asynchronous |
| Form custom validation | Defined as Functions | Defined as Directives |
| Testing | No interaction with change detection cycle | Need knowledge of the change detection process |
| Mutability | Immutable(by always returning new value for FormControl instance) | Mutable(Property always modified to new value) |
| Scalability | More scalable using low-level APIs | Less scalable using due to abstraction on APIs |

1. **What are the state CSS classes provided by ngModel?**

The ngModel directive updates the form control with special Angular CSS classes to reflect it's state. Let's find the list of classes in a tabular format,

| **Form control state** | **If true** | **If false** |
| --- | --- | --- |
| Visited | ng-touched | ng-untouched |
| Value has changed | ng-dirty | ng-pristine |
| Value is valid | ng-valid | ng-invalid |

### How to set ngFor and ngIf on the same element?

Sometimes you may need to both ngFor and ngIf on the same element but unfortunately you are going to encounter below template error.

Template parse errors: Can't have multiple template bindings on one element.

In this case, You need to use either ng-container or ng-template. Let's say if you try to loop over the items only when the items are available, the below code throws an error in the browser

<ul \*ngIf="items" \*ngFor="let item of items">

<li></li>

</ul>

and it can be fixed by

<ng-container \*ngIf="items">

<ul \*ngFor="let item of items">

<li></li>

</ul>

</ng-container>