**1. What is Angular 4 and how it differs from Angular 1.x?**

Angular 4 is a Javascript framework built around the concept of components, and more precisely, with the Web Components standard in mind. It was rewritten from scratch by the Angular team using Typescript (although we can use it with *ES5*, *ES6*, or *Dart* as well).

Angular 4 is a big change for us compared to 1.x. Because it is a completely different framework than 1.x, and is not backward-compatible. Angular 4 is written entirely in Typescript and meets the ECMAScript 6 specification. The main differences are:

* Angular 4 is entirely component based. **Controllers** and **$scope** are no longer used. They have been replaced by components and directives.
* Angular 4 uses TypeScript. TypeScript will not be used in the browser directly. So the program code is compiled to JavaScript. This can be achieved with “Traceur”.
* The digest cycle from Angular 1.x has been replaced by another internal mechanism known as “**Change Detection**”. This feature, along with other improvements and tweaks, yields a considerable increase in performance.
* Unlike Angular 1.x where we can get most of the functionalities in angular.js file, Angular 4 follows module pattern. We need to import the functions ourself and export them when we need anywhere else.
* There are no more **factory**, **service**, **provider** in Angular 4. We need to use **class** for declaring a service.

**2. What is Component in Angular 4? How do you declare them?**

In Angular 4, everything is a component. A component controls a patch of screen called a *view*. Components are the most basic building block of an UI in an Angular application. An Angular application is a tree of Angular components. Unlike directives, components always have a template and only one component can be instantiated per an element in a template.

To declare a component we must first import Component from **@angular/core** library. And then, define a *metadata* for the component, where we can configure our component. A simple component would look like below:

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

@Component({

selector: 'simple-component',

template: `

<div>

<p>This is a simple component in Angular 4 with a message {{message}}.</p>

</div>

`

})

export class SimpleComponent {

message: string;

constructor() {

this.message = 'Welcome'

}

}

To include the component in the view, we use **<simple-component></simple-component>**.

Angular 4 components make some decisions for you that directives didn't. For example, Angular 4 components:

* Always isolate scope (instead of sharing a parent scope)
* Always restrict **“E”** (load into custom elements in the DOM)
* Always bind to a controller (as opposed to **$scope**)

**3. What is component decorators in Angular 4?**

The main objectives of decorators is to add some metadata to the class that will tell Angular 4 how to process a class. Or in another words, Decorators are functions that modify JavaScript classes. Angular has many decorators that attach metadata to classes so that it knows what those classes mean and how they should work.

If we consider Component in Angular 4, we will have following options to configure.

* **selector:** - define the name of the HTML element in which our component will live.
* **template** or **templateUrl:** - It can be inline string or link an external html file. It allows us to tie logic from our component directly to a view.
* **styles:** - the styles array for our specific component. We can also link external CSS by **styleUrls**.
* **directives:** - another component directives we want to use inside our components.
* **providers:** - This is the place we are passing the services that we need insider our components.

Immediately after this decorator or right to it, we need to export a class where our variables and functions reside that our component uses.

export class SampleComponent {

name: 'Angular 4';

}

**4. What is compilation in Angular 4? And what are the types of compilation in Angular 4?**

An Angular application consists largely of components and their HTML templates. Before the browser can render the application, the components and templates must be converted to executable JavaScript by the *Angular compiler*.

There is actually only one Angular compiler. The difference between AOT and JIT is a matter of timing and tooling. There are two types of compilation Angular 4 provides.

* **Just-in-time (JIT) compilation:** This is a standard development approach which compiles our Typescript and html files in the browser at runtime, as the application loads. It is great but has disadvantages. Views take longer to render because of the in-browser compilation step. App size increases as it contains angular compiler and other library code that won't actually need.
* **Ahead-of-time (AOT) compilation:** With AOT, the compiler runs at the build time and the browser downloads only the pre compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first. This compilation is better than JIT because of Fast rendering, smaller application size, security and detect template errors earlier.

**5. What is <code>@NgModule</code>?**

An **NgModule** class describes how the application parts fit together. Every application has at least one **NgModule**, the root module that we bootstrap to launch the application.

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

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

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

@NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Here the **AppComponent** is the root module of our application that Angular creates and inserts it into the **index.html** page.

**6. What are all the <i>metadata</i> properties of <code>NgModule</code>? And what are they used for?**

**@NgModule** accepts a metadata object that tells Angular how to compile and launch the application. The properties are:

* **imports** – Modules that the application needs or depends on to run like, the **BrowserModule** that every application needs to run in a browser.
* **declarations** – the application's components, which belongs to the **NgModule** class. We must declare every component in an **NgModule** class. If we use a component without declaring it, we'll see a clear error message in the browser console.
* **bootstrap** – the root component that Angular creates and inserts into the **index.html** host web page. The application will be launched by creating the components listed in this array.

**7. How do you bootstrap the Angular 4 application?**

There are many ways to bootstrap an application. The recommended place to bootstrap a JIT-compiled browser application is in a separate file in the **src** folder named **src/main.ts**

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

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);

This code creates a browser platform for dynamic (JIT) compilation and bootstraps the **AppModule** described above.

In the AOT compilation, the Angular compiler runs ahead of time as part of the build process, producing a collection of class factories in their own files. Among them is the **AppModuleNgFactory**. The syntax for bootstrapping the pre-compiled **AppModuleNgFactory** is as follow.

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

*// The app module factory produced by the static offline compiler*

import { AppModuleNgFactory } from './app/app.module.ngfactory';

*// Launch with the app module factory.*

platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);

**8. What is Template reference variables?**

A template reference variable (**#var**) is a reference to a DOM element within a template. We use hash symbol (#) to declare a reference variable in a template.

<input #name placeholder="Your name">

{{ name.value }}

In the above code the **#name** declares a variable on the **input** element. Here the **name** refers to the ***input*** element. Now we can access any property of the **input** DOM, using this reference variable. For example, we can get the value of the **input** element as **name.value** and the value of the placeholder property by **name.placeholder** anywhere in the template.

Finally, a Template reference variable refers to its attached element, component or directive. It can be accessed anywhere in the entire template. We can also use **ref-** instead of #. Thus we can also write the above code as **ref-name**.

**9. What is Template input variables?**

Sometimes we will create a local variable in a template, and those variables are called Template input variables. For example,

<div \*ngFor="let item of items">{{item.name}}</div>

Here the **let** keyword before **item** creates a template *input variable* called **item**. The **ngFor** directive iterates over the **items** array and sets **item** to the current item from the array during each iteration. It is equivalent to **ng-repeat** in Angular 1.x.

A template input variable is not the same as a template reference variable. A template input variable is a variable whose value you can reference within a *single* instance of the template. But a tempalte reference variable can be accessed *anywhere* in the entire template.

**10. What are structural directives?**

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements. Structural directives are easy to recognize. An asterisk (\*) precedes the directive attribute name as in this example.

<ul>

<li \*ngFor="let name of names">{{name}}</li>

</ul>

The **ngFor** directive iterates over the component's **names** array and renders an instance of this template for each **name** in that array.

Some of the other structural directives in Angular are **ngIf** and **ngSwitch**.

**11. What does asterisk (\*) syntax means in the structural directives?**

The asterisk is "syntactic sugar" for something a bit more complicated. Internally, Angular desugars it in two stages. Consider the below code:

<div \*ngIf="user">{{user.name}}</div>

First, Angular translates the **\*ngIf="..."** into a template attribute, **template="ngIf ..."**, like this.

<div template="ngIf user">{{user.name}}</div>

Then it translates the template attribute into a **<ng-template>** element, wrapped around the host element, like this.

<ng-template [ngIf]="user">

<div>{{user.name}}</div>

<ng-template>

**12. What is <code>&lt;ng-template&gt;</code>?**

The **<ng-template>** is an Angular element for rendering HTML. It is never displayed directly. In fact, before rendering the view, Angular replaces the **<ng-template>** and its contents with a comment.

<div \*ngIf="user">{{user.name}}</div>

In the above code, the directive will compiled as mentioned in the previous answer and then finally append to DOM as follow:

*<!--template bindings={*

*"ng-reflect-ng-if=": "[object object]"*

*}-->*

<div ng-content-exn-0>Angular 4 User</div>

**13. What is Component lifecycle?**

A Component/Directive in Angular has lifecycle as Angular creates, updates and destroy them. We can tap into the key moments in that lifecycle and implement lifecycle hook interfaces in the Angular library.

Angular creates a component/directive by calling its constructor. After that Angular calls the lifecycle hook in the below order:

* **ngOnChanges()** - calls when Angular sets or re-sets the data-bound properties. Called before **ngOnInit()** and whenever one or more data-bound input properties change.
* **ngOnInit()** - called when Angular initializes the Component. It is called only once.
* **ngDoCheck()** - Detect and act upon changes that Angular can't or won't detect on its own. It is called immediately after **ngOnChanges()** and **ngOnInit()**.
* **ngAfterContentInit()** - Respond after Angular projects external content into the component's view. It is a Component only hook and can not be called for a directive.
* **ngAfterContentChecked()** - Respond after Angular checks the content projected into the component. Called after the **ngAfterContentInit()**. It will also be called every subsequent **ngDoCheck()**. This is a Component-only hook.
* **ngAfterViewInit()** - Respond after Angular initializes the component's views and child views. This is a Component-only hook.
* **ngAfterViewChecked()** - Respond after Angular checks the component's views and child views. A component-only hook.
* **ngOnDestroy()** - Called just before Angular destroys the directive/component. This is the place we can unsubscribe observables and detach event handlers to avoid memory leaks.

export class AppComponent implements OnInit {

constructor() { }

*// implement OnInit's `ngOnInit` method*

ngOnInit() { console.log(`OnInit`); }

}

**14. What is the <i>expression context</i> in Angular 4?**

The *expression context* is typically the component instance. When we bind any value within double curly braces or in quotes, then it refers to the properties of the Class instance of that Component.

{{ name }}

<span [hidden]="isValid">Valid</span>

An expression may also refer to properties of the *template's* context such as a template input variable or a template reference variable. For example, in the below snippet, the **let name** refers to the *template input variable* and **#name** refers to the *template reference variable*.

<li \*ngFor="let name of names">{{name}}</li>

<input #name placeholder="Your name">

{{ name.value }}

The context for terms in an expression is a combination of the template variables, the directive's context object, and the component's members. If we reference a name that belongs to more than one of these namespaces, the template variable name takes precedence, followed by a name in the directive's context, and, lastly, the component's member names.

In the previous example, the **{{name}}** refers to the template input variable **names** not the component's property **name** even it has one declared.

**15. What are all the <i>binding categories</i> in Angular?**

Binding types in Angular 4 are grouped into three categories distinguished by the direction of data flow: in one-way data binding: from the source-to-view, from view-to-source, and in two-way: view-to-source-to-view.

To bind a data from source to view, we can use interpolation **{{ }}** and attribute binding.

{{expression}}

[target]="expression"

bind-target="expression"

For data flow from view to source, we can use event binding.

(target)="statement"

on-target="statement"

For two-way data flow, we can use **[( )]** syntax, which is also called *banana-in-the-box*

[(target)]="statement"

bindon-target="statement"

**16. Explain the different types of bindings available in Angular?**

* **Property binding** - This most common binding is used to set property of an DOM element through expression. This is a one-way data binding which means you can not read the value from this property. We can *only* set it.

**<img [src]="imageUrl">**

* **Event binding** - This is also one-way data binding but the data flows from view to source. It binds an expression to the given element on DOM element and Components.

**<button (click)="onSubmit()">Submit</button>**

* **Two-way binding** - Binds both event and property.

**<input [(ngModel)]="name"/>**

* **Attribute binding** - Set the value of an attribute on a DOM directly. We must use this attribute binding when there is no element property to bind. For example, consider the ARIA, SVG and table span attributes. They are not property targets. If we use property bindings on them, we will get error.
* **<tr><**
* **td [attr.colspan]="1 + 1">One-Two</td>**

**</tr>**

* **Class binding** - Class binding syntax resembles property binding. Instead of an element property between brackets, they start with the prefix class, optionally followed by a dot (**.**) and the name of a CSS class: **[class.class-name]**. If the class is already present in the class list then Angular removes it.

**<div class="header" [class.special]="isSpecial">This is a class binding.</div>**

* **Style binding** - We can set inline styles with a style binding. The syntax is same as **class binding** except the prefix which **style** instead of **class**. We can also use unit extension at the end.
* **<h1 [style.background-color]="isRed ? 'red': 'black'">Heading 1</h1>**

**<h1 [style.font-size.em]="isRed ? 2: 1">Heading 1</h1>**

**17. What is Directive in Angular 4? How it differs from Components?**

Directives allow us to attach behavior to elements in the DOM, for example, doing something on mouse over or click. In Angular, a Directive decoraor (**@Directive**) is used to mark a class as an Angular directive and provides additional metadata that determines how the directive should be processed. Below are the metadata properties of a directive.

* **selector** - css selector that identifies this component in a template
* **host** - map of class property to host element bindings for events, properties and attributes
* **inputs** - list of class property names to data-bind as component inputs
* **outputs** - list of class property names that expose output events that others can subscribe to
* **providers** - list of providers available to this component and its children
* **queries** - configure queries that can be injected into the component
* **exportAs** - name under which the component instance is exported in a template

A Component is a directive with a template. So we should use a Component whenever we want reusable set of DOM elements with behaviors of UI. And we should use a Directive whenever we want reusable behavior to supplement the DOM.

**18. What are all the types of Directives?**

There are three types of directives in Angular. They are **attribute directives**, **structural directives**, and **components**.

* **Structural directives** change the DOM layout by adding and removing DOM elements. For example, **\*ngIf** and **\*ngFor**
* **Attribute directives** change the appearance or behavior of an element. . For example, **\*ngStyle** and **\*ngClass**
* **Components** are basically directives with a template.

**19. How do components communicate with each other?**

There are no **$rootScope**, **$broadCast** or **$emit** in Angular 4. But Angular follows new style of communication via **EventEmitters**.

* **Inputs** allow us to pass data *down* to a child Component.
* **Outputs** allows to pass data *up* to a parent Component.

import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({

selector: 'sample-component',

templateUrl: 'sample.component.html'

})

export class SampleComponent {

constructor() {}

@Input() title: string;

@Output() onTitleChanged = new EventEmitter();

changeName(newName: string) {

this.onNameChanged.emit(newName);

}

}

We pass the values down to a Child Component via property and set the value in Child Component using **@Input** aliases

To communicate the Parent Component, the child exposes an **EventEmitter** property with which it calls **emit** events. The parent binds to that event property and reacts to those events.

When we need the Parent Component to read or write the Child component's values or call methods, we should *inject* the Child component into the parent as a *ViewChild*.

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

import { CountdownTimerComponent } from './a-child.component';

@Component({

selector: 'parent-component',

template: `<h3>Parent Component</h3><a-child></a-child>`

})

export class ParentComponent{

@ViewChild(AChildComponent)

private aChildComponent: AChildComponent;

}

*// Now we can read or write child component property or call a method using this.aChildComponent.*

Also, a parent and child component can share a service to enables bi-directional communication.

**20. What are the difference between <code>Renderer</code> and <code>ElementRef</code> in Angular 4?**

The **Renderer** in Angular is a class that is an abstraction over the DOM. Using the **Renderer** for manipulating the DOM doesn't break server-side rendering or Web Workers (where direct access to the DOM would break).

For example, if we want to focus an input element, we could use template reference variable and with the help of **@ViewChild** decorator we can access the native DOM. But we are actually depending the native DOM and lose the opportunity to be able to run our script also in no-DOM environments like native mobile or Web workers. Remember that Angular is a platform, and the browser is just one option for where we can render our app.

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

@Directive({

selector: '[exploreRenderer]'

})

export class ExploreRendererDirective {

private nativeElement : Node;

constructor( private renderer : Renderer, private element : ElementRef ) {

this.nativeElement = element.nativeElement;

}

}

*//Whereever we want to use the native DOM, we could use this directive on the element like <div exploreRenderer></div>*

**ElementRef** is a class that can hold a reference to a DOM element. This is again an abstraction to not break the environments where the browsers DOM isn't actually available. If **ElementRef** is injected to a component, the injected instance is a reference to the host element of the current component.

There are other ways to acquire an **ElementRef** instance like **@ViewChild()**, **@ViewChildren()**, **@ContentChild()**, **@ContentChildren()**. In this case **ElementRef** is a reference to the matching element(s) in the template or children.

**Renderer** and **ElementRef** are not "either this or that", but instead they have to be used together to get full platform abstraction.

**Renderer** acts on the DOM and **ElementRef** is a reference to an element in the DOM the Renderer acts on.

**21. What is an Angular 4 Services?**

Just like Angular 1.x, Services in Angular 4 are Javascript functions along with its associated properties and methods. A service can be injected into components via Dependency Injection.

A Service class should be decorated with **@Injectable()** decorator and should be included in the **providers** array in **NgModule**.

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

@Injectable()

export class MyService {

}

To use the service in one of the Component, we muse include it in the **constructor** functions parameters.

constructor(private \_myService: MyService) {

}

Services can be included the Components level **Providers**. But by doing so, the service will be available only to its child components. In order to avail the service in other components, we need to inject them again in the another component. As services are singleton, injecting them in two places creates two instances of the service. So it is best to inject the service in **NgModule** and use it application level if the service is needed in different places.

**22. What are all the uses of a service?**

Services encapsulates business logic and separates them from UI concerns or the controller concerns, which governs them both.

Services focus on functionality thus benefits in maintainability. The separation of UI logic from business logic is intended to reduce the coupling between the UI layer and the Model layer, leading to a cleaner design that is easier to develop, test, and maintain.

**23. What is the difference between <code>@Inject</code> and <code>@Injectable</code>?**

**@Inject()** is a manual mechanism for letting Angular know that a parameter must be injected. It can be used like so:

constructor(@Inject(MyService) private \_myService) {

}

The **MyService** will be instantiated behind the scenes and is injected in the constructor by Angular.

But when using TypeScript, **@Inject** is only needed for injecting primitives. TypeScript's types let Angular know what to do in most cases. The above example would be simplified in TypeScript to:

constructor(private \_myService: MyService) {

}

On the other hand, **@Injectable()** lets Angular know that a class can be used with the dependency injector. **@Injectable()** is not strictly required if the class has other Angular decorators on it or does not have any dependencies. What is important is that any class that is going to be injected with Angular is decorated. However, best practice is to decorate injectables with **@Injectable()**, as it makes more sense to the reader.

**24. What is <i>Pipe</i> in Angular?**

Pipes are equivalent of filters in Angular 1.x. Pipes are functions that returns a value and are used to modify or format a value in the template.

Same as Angular 1.x, a pipe start with the pipe (**|**) symbol after the value and any paramter should be seprated with the colon (**:**) symbol. We can also chain multiple pipes.

myValue | myPipe:param1:param2 | mySecondPipe:param1

To create a custom pipe, we should decorate a class with **@Pipe** decorator.

import { Pipe, PipeTransform } from 'angular2/core';

@Pipe({ name : 'removeSpaces'

})

export class RemoveSpaces implements PipeTransform { transform(value){

return value.replace(/ /g, '');

}

}

The above Pipe class implements the **PipeTransform** interface's transform method. This method takes the value being piped and a variable number of arguments of any type and return a transformed ("piped") value.

**25. What is Pure and Impure Pipes?**

**Pure pipes** are stateless that flow input date without remembering anything or causing detectable side-effects. Pipes are pure by default, hence most pipes are pure. We can make a pipe impure by setting its pure flag to **false**. Angular executes a pure pipe only when it detects a *pure* change to the input value. A pure change is either a change to a primitive input value or a changed object reference.

**Impure pipes** are those which can manage the state of the data they transform. A pipe that creates an HTTP request, stores the response and displays the output, is a impure or stateful pipe. Stateful Pipes should be used cautiously. Angular provides **AsyncPipe**, which is stateful. In the following code, the pipe only calls the server when the request URL changes and it caches the server response. The code uses the Angular http client to retrieve data:

@Pipe({

name: 'fetch',

pure: false

})

export class FetchJsonPipe implements PipeTransform {

private cachedData: any = null;

private cachedUrl = '';

constructor(private http: Http) { }

transform(url: string): any {

if (url !== this.cachedUrl) {

this.cachedData = null;

this.cachedUrl = url;

this.http.get(url)

.map( result => result.json() )

.subscribe( result => this.cachedData = result );

}

return this.cachedData;

}

}

**26. What is Lazy loading?**

Lazy loading is the act of only loading parts of our app when they are needed. Lazy loading is one of the features we can add to our app to improve performance.

In the following code, we have two routes: **home** and **lazy**. By the default, the application is redirected to **home** and **HomeComponent** is called. But when we go to **lazy** path, we are trying to load a Module (**LazyModule**) dynamically

const routes: Routes = [

{ path: '', redirectTo: 'home', pathMatch: 'full' },

{ path: 'home', component: HomeComponent },

{ path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }

];

Instead of **Component**, we have used **loadChildren** property to load the specified module. The lazy module file will contain a Angular module and exports the **LazyModule** class as shown below:

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

import { LazyComponent } from './lazy.component';

import { routing } from './lazy.routing';

@NgModule({

imports: [routing],

declarations: [LazyComponent]

})

export class LazyModule {}

If we define the class **LazyModule** as the default export of the file, we don't need to define the class name in the **loadChildren** property as shown above.

Until the app navigated to **lazy**, the **LazyModule** has not being downloaded.

**27. What is View Encapsulation?**

View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:

* **ViewEncapsulation.Emulated** - This is default. The styles from the main HTML propagate to this component and styles defined in this components decorator are scoped to this component only. It is not using Native Shadow DOM. But to encapsulate the styles within the components, Angular adds some attributes like **\_ngcontent-1**, **\_ngcontent-2** or **\_nghost-1** with the css rules like **.header[\_ngcontent-1] {}** to make sure the styles scoped and applied to the particular component only.
* **ViewEncapsulation.Native** - The styles from the main HTML do not propagate to this component and styles defined in this components decorator are scoped to this component only. This one uses native Shadow DOM encapsulation.
* **ViewEncapsulation.None** - The styles from the component propagate back to the main HTML and therefore are visible to all components on the page. All components with **None** encapsulation will have their styles duplicated in all components with **Native** encapsulation. This also means that all the styles apply to the entire document. Or in other words, a component could overwrite styles from another component because its styles are applied to the document head later. That's why this is the *unscoped* strategy.

@Component({

*// ...*

encapsulation: ViewEncapsulation.None,

styles: [*// ...]*

})

export class SampleComponent {

*// ...*

}

**28. What is Shadow DOM in Angular 4? Does Angular uses Virtual DOM?**

Shadow DOM provides encapsulation for DOM and CSS in a Component. When we create a component in Angular, Angular puts its template into a **shadowRoot** (not by default, but by using **ViewEncapsulation.Native**), which is the Shadow DOM of the particular component. Below is the example DOM structure of a Shadow DOM.

<my-component title="Header">

#shadow-root

| <style>

| .head {

| background: blue;

| }

| </style>

| <div class="head">

| <div (click)="toggle()" class="header\_\_title">

| Details

| </div>

| <div class="header\_\_content">

| <content></content>

| </div>

| </div>

"This is a Head"

</my-component>

Doing so we can get the encapsulation of DOM and styles. But if we want to use our component in No-DOM environment or Not-supported browsers, we can use **ViewEncapsulation.Emulated** for emulation.

Shadow DOM and Virtual DOM are two different things. And Shadow DOM is not targeting performance as virtual DOM is, but style encapsulation. And Angular 4 doesn't use **virtual DOM** at all. Angular 4 has change detection that detects changes to the model and only updates the parts of the DOM that need to be changed according to the model changes.

**29. Explain Angular Routing.**

We can configure our application routes as array of objects and pass it to the Angular's **RouterModule.forRoot**.

*//app/app.routes.ts*

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

const routes: Routes = [

{ path: 'component-one', component: ComponentOne }, { path: 'component-two', component: ComponentTwo }

];

export const routing = RouterModule.forRoot(routes);

We then import our routing configuration in the root of our application.

*//app/app.module.ts*

import { routing } from './app.routes';

@NgModule({

imports: [ BrowserModule,

routing

],

declarations: [

AppComponent,

ComponentOne,

ComponentTwo

],

bootstrap: [ AppComponent ]

})

export class AppModule {}

By default, the application navigates to the empty route. We can redirect it to one of the named route.

{ path: '', redirectTo: 'component-one', pathMatch: 'full' },

The **pathMatch** property, which is required for redirects, tells the router how it should match the URL provided in order to redirect to the specified route. Since **pathMatch: full** is provided, the router will redirect to **component-one** if the entire URL matches the empty path (**''**).

**30. How to link routes in HTML?**

To add link to one of the routes, we should use **RouterLink** directive in HTML. This directive accepts an array. The first param is the name of the route and the second param is the parameters that we want to pass with the route.

<a [routerLink]="['component-one', routeParams]"></a>

If we use the **routerLink** directive without the brackets, we need to the pass the route as a string.

<a routerLink="/component-one"></a>

We can also change the route programmatically in Angular.

this.router.navigate(['/component-one']);

The **<router-outlet></router-outlet>** acts as a placeholder for component. Angular dynamically adds the component for the route being activated into this.

**31. How do we access the Route parameters in Angular?**

The router provides two different ways to grab the route parameters. They are:

**The Snapshot Way:** The **ActivatedRoute** service provides us with a snapshot of the current route.

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

import { ActivatedRoute } from '@angular/router';

@Component({

..

})

export class SampleComponent implements OnInit {

constructor(private route: ActivatedRoute) {}

ngOnInit() {

this.myParam = this.route.snapshot.params.myParam;

}

This is simple and enough. But since Angular reuse the components for performance, this method won't work for some cases. For example, if we navigate from **/products/1** to **/products/2**. In that cases we need to use another way which is **The Observable/Stream Way**.

**The Observable/Stream Way:** This uses observables and when we navigate from one route other route, the Observable will pass along the new data. Angular v4+ router provides us with ParamMap which is an Observable we can use.

ngOnInit() {

*// subscribe to the parameters observable*

this.route.paramMap.subscribe(params => {

console.log(params.get('myParam'));

this.myParam = params.get('myParam');

});

}

If we are working with Angular versions prior to v4, we can grab route parameters with **params** instead of **paramMap**.

ngOnInit() {

*// subscribe to the parameters observable*

this.route.params.subscribe(params => {

console.log(params.get('myParam'));

this.myParam = params.get('myParam');

});

}

**32. How and when do you define a Child route?**

When some routes may only be accessible and viewed within other routes it may be appropriate to create them as child routes. For example, if we have a product route with individual id and we want to navigate to other routes to see its overview or specification.

The **children** property in the routes accepts an array of child routes. And we can use the same **<router-outlet>** directive to hold the child component.

export const routes: Routes = [

..

{ path: 'product-details/:id', component: ProductDetails,

children: [

{ path: '', redirectTo: 'overview', pathMatch: 'full' },

{ path: 'overview', component: Overview },

{ path: 'specification', component: Specification }

]

}

];

**33. What is the difference between <code>queryParams</code> and <code>routeParams</code>?**

The key difference between query parameters and route parameters is that route parameters are essential to determining route, whereas query parameters are optional. The optional parameters are equal to **/product-list?page=2** where as route params are **/product-list/2**.

//Pass query params in router link

<a [routerLink]="['product-list']" [queryParams]="{ page: 2 }">Go to Page 2</a>

*//query parameters programmatically*

this.router.navigate(['/product-list'], { queryParams: { page: pageNum } });

To read the query paramters, we need to subscribe to **queryParams** in the **ActivatedRoute**.

ngOnInit() {

this.sub = this.route.queryParams.subscribe(params => {

this.page = +params['page'] || 0;

});

}

**34. How to restrict or control access from or to a Route? What is Route Guards?**

To control whether the user can navigate to or away from a given route, we should use **Route Guards**. Route guards can also control whether a user can leave a certain route. For example, say the user has typed information into a form on the page, but has not submitted the form. If they were to leave the page, they would lose the information. We may want to prompt the user if the user attempts to leave the route without submitting or saving the information.

In order to use route guards, we must register them with the specific routes we want them to run for. In our route config we can add our guards to that route as follows:

const routes: Routes = [

{

path: 'restricted',

component: AccountPage,

canActivate: [ AuthRouteGuard ],

canDeactivate: [ SaveFormsGuard ]

}

Now **AuthRouteGuard** will be checked by the router when activating the accounts route, and **SaveFormsGuard** will be checked when leaving that route. The following code implements the **canActivate** interface.

import { CanActivate } from '@angular/router';

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

@Injectable()

export class AuthRouteGuard implements CanActivate {

constructor() {}

canActivate() {

*//check the user and return true of false here.*

}

}

If **canActivate** returns true, the user can activate the route. Otherwise the user cannot access the route. Import and implement **CanDeactivate** interface to safely navigate from the route.

The **canActivate** and **canDeactivate** functions can either return values of type **boolean**, or **Observable<boolean>** (an Observable that resolves to boolean). If we need to do an asynchronous request (like a server request) to determine whether the user can navigate to or away from the route, we can simply return an **Observable<boolean>**. The router will wait until it is resolved and use that value to determine access.

**35. What is Auxilary Routes?**

These are just plain routes like the primary route that was mapped to the router-outlet component. But instead, auxiliary routes are mapped to a different outlet which must be named (unlike the primary outlet).

Angular supports the concept of auxiliary routes, which allow you to set up and navigate multiple independent routes in a single app. Each component has one primary route and zero or more auxiliary outlets. Auxiliary outlets must have unique name within a component. Following code is how we configure auxiliary route.

//In HTML

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

export const routeConfig:Routes = [

{

path: 'main',

....

},

{

path: 'sidebar',

outlet: 'aside',

component: Sidebar

}

];

To trigger a auxiliary route in HTML router link:

<a [routerLink]="[{ outlets: { 'aside': ['aside'] } }]">Auxiliary Route>/a>

Auxiliary route introduces special syntax to define auxiliary route URLs in the same Primary URL. The URL would look like this: **/main(aside:sidebar)**

**36. What is Redux and <code>@ngRx</code>?**

Redux is an application state manager for JavaScript applications, and keeps with the core principles of the Flux-architecture by having a unidirectional data flow in our application. Redux applications have only one global, read-only application state. This state is calculated by "reducing" over a collection or stream of actions that update it in controlled ways.

**@ngrx** is a set of modules that implement the same way of managing state as well as some of the middleware and tools in the Redux ecosystem. In other way, **ngrx** is a collection of reactive libraries for angular, containing a redux implementation and many other useful libraries.

Using this technique, we keep our application state in Store and everything saved in the store is read only. The only way to change the state is to emit an action, an object describing what happened.

**37. What is Action in Redux?**

Redux uses a concept called Actions, which describe state changes to our application. Redux actions are simple JSON objects that implement the Action interface provided by **@ngrx**.

export interface Action {

type: string;

payload?: any;

}

The **type** property is a string used to uniquely identify your action to your application. The **payload** property provides a way to pass additional data to other parts of Redux, and it's entirely optional. The following is an example creation of an Action:

const exampleAction: Action = {

type: 'EXAMPLE\_ACTION',

payload: {

name: 'Angular',

}

};

If we want to change the state of our application anyway, we have to make a request in the form of an action.

Overall, we can think of actions in Redux as events. They report that something has happened. Thats why actions do not describe how the state has changed. Instead they precisely describe what happened.

**38. How do we change the state of our application? What is Reducers?**

Reducers are like listeners that are listening for some actions. These reducers are the only structure that is allowed to alter the state of our application. All state changes are implemented inside of reducers.

After all, a reducer is just a function. It takes the current state and the action as parameters. All we need to do, is to create a function with that interface, and register it at our store. Here is the sample reducer function.

export function reducer(state: number = 1, action: amount.AmountChangeAction) {

switch (action.type) {

case amount.AMOUNTCHANGE:

*//return modified state*

default:

return state;

}

}

**39. What is Side-effects? How to handle them?**

Reducer should be pure function which means they should not depend on anything other than its own scope. Hence it is side-effect free. But often times, it is not possible and we need to handle these side-effects.

The most common example of side-effect is HTTP requests. We are sending a request and not sure when it comes back. It is an asynchronous operation. To deal with this kind of side-effects, the **ngrx** team has created a module called **ngrx/effects**.

All we need to do, is to use the @Effect() decorator to decorate a method of an @Injectable. **@ngrx/effects** provides an Angular **Action** service (which is also an **Observable**) to emit every action that has been dispatched by your application in a single stream. Its **ofType()** method can be used to filter the one or more actions we're interesting in before adding a side-effect. Regardless if this requests succeeds or fails, we need to create and return an **Observable** that is bound to the new action we'd like Redux to perform

constructor( private \_actions: Actions, private apiService: ApiService ) {}

@Effect()

login = this.\_actions

.ofType(actions.USER\_LOGIN)

.switchMap(action => this.apiService.login(action.payload)

.map(result => createAction(actions.RETRIEVE\_SUCCESS, result.json()))

.catch(error => Observable.of(createAction(actions.RETRIEVE\_ERROR, error.json())))

);

**40. How to prevent security threads in Angular App? What are all the ways we could secure our App?**

Some of them are:

* Avoid using/injecting dynamic HTML content to your component.
* If using external HTML which is coming from database or somewhere outside the application, sanitize it before using.
* Try not to put external urls in the application unless it is trusted. Avoid url re-direction unless it is trusted.
* Consider using AOT compilation or offline compilation.
* Try to prevent XSRF attack by restricting the api and use of the app for known or secure environment/browsers.

**41. How to optimize Angular app?**

* Consider lazy loading instead of fully bundled app if the app size is more.
* Make sure that any 3rd party library, which is not used, is removed from the application.
* Have all dependencies and dev-dependencies are clearly separated.
* Make sure the application doesn't have un-necessary import statements.
* Make sure the application is bundled, uglified, and tree shaking is done.
* Consider AOT compilation.

**42. What is Change Detection in Angular? How it is improved from Angular 1.x?**

Change detection is the process that allows Angular to keep our views in sync with our models. In Angular 1, we have two-way data bindings and have a long list of watchers that needed to be checked every-time a digest cycle was started. This was called *dirty checking* and it was the only change detection mechanism available.

In Angular, the flow of information is **unidirectional**, even when using ngModel to implement two way data binding. In this new version, our code is responsible for updating the models. Angular is only responsible for reflecting those changes in the components and the DOM by means of the selected change detection strategy.

By default, Angular defines a certain change detection strategy for every component in our application. To make this definition explicit, we can use the property changeDetection of the **@Component** decorator. Angular provides two Change Detection Strategies:

* **ChangeDetectionStrategy.Default** - The default strategy for the change detection is to traverse all the components of the tree even if they do not seem to have been modified.
* **ChangeDetectionStrategy.OnPush** - This instructs Angular to run change detection on these components and their sub-tree only when new references are passed to them versus.

Below is the example code for default change detection:

*//app.component.ts*

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent {

fruits = ['Banana', 'Apple', 'Orange'];

addFruit(fruit) {

this.fruits.push(fruit);

}

}

And the template:

<input #newFruit type="text" placeholder="Enter a fruit">

<button (click)="addFruit(newFruit.value)">Add fruit</button>

<app-child [items]="fruits"></app-child>

The child component,

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

@Component({

selector: 'app-child',

templateUrl: './child.component.html'

})

export class ChildComponent {

@Input() items: string[];

}

Child component's template:

<ul>

<li \*ngFor="let item of items">{{ item }}</li>

</ul>

Angular creates change detector classes at runtime for each component, and keep track of the previous values to check with the new value whenever a change is detected. If we add a fruit in the parent component it will get reflected in the child immediately. The **OnPush** strategy is different that it will **only** run the change detection when the component's state is **mutated**. The same child component with this strategy is as follow:

import { Component, Input, **ChangeDetectionStrategy** } from '@angular/core';

@Component({

selector: 'app-child',

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

changeDetection: **ChangeDetectionStrategy.OnPush**

})

export class ChildComponent {

@Input() items: string[];

}

Now if we add a fruit in the parent component, it will not be updated in the child. Because the child component is receiving the same array reference every time. In order to make this work, we need to pass the new reference of **fruits** array every time we change it.

*//app.component.ts*

addFruit(fruit) {

this.foods = [...this.fruits, fruit];

}

Now everything is working fine because Angular able to identify new reference and can run the change detection. **OnPush** change detection is very useful when our application has lot of components as it will boost the application performance.

**43. What is <code>ChangeDetectorRef</code>?**

This class provides five methods that we can use to get more control over the Change Detection run in Angular.

**detectChanges()** - This can be used to implement local change detection checks and its children. For example, considering the previous example, instead of passing new references, we can keep adding fruits in the parent component and have a refresh button in the child to detect changes.

constructor(private cd: ChangeDetectorRef) {}

refresh() {

this.cd.detectChanges();

}

**markForCheck()** - Consider our previous example, but if we have to use *Observables* for getting fruits data. We are subscribing to it in the **OnInit** life hook in our child component.

@Input() items: Observable;

fruits: string[] = [];

ngOnInit() {

this.items.subscribe(fruit => {

this.fruits = [...this.fruits, ...fruit];

**this.cd.markForCheck();**

});

}

Here the new data mutates our items observable, so Angular doesn't run change detection. **markForCheck** instructs Angular that this particular input should trigger change detection when mutated.

**detach() and reattach()** - **detach()** is used to detache the change detector from the change detector tree. The detached change detector will not be checked until it is reattached. **reattach()**, as the name suggests, reattach the change detector to the change detector tree. This also marks **OnPush** ancestors as to be checked.

**checkNoChanges()** - It checks the change detector and its children, and throws if any changes are detected. This is used in development mode to verify that running change detection doesn't introduce other changes.

**44. What is <code>NgZone</code> service? How Angular is notified about the changes?**

**Zone.js** is one of the Angular dependencies which provides a mechanism, called zones, for encapsulating and intercepting asynchronous activities in the browser (e.g. **setTimeout**, **setInterval**, **promises**). These zones are *execution contexts* that allow Angular to track the start and completion of asynchronous activities and perform tasks as required (e.g. change detection). Zone.js provides a global zone that can be forked and extended to further encapsulate/isolate asynchronous behaviour, which Angular does so in its **NgZone** service, by creating a fork and extending it with its own behaviours.

The **NgZone** service provides us with a number of Observables and methods for determining the state of Angular's zone and to execute code in different ways inside and outside Angular's zone.

**NgZone** exposes a set of Observables that allow us to determine the current status, or stability, of Angular's zone.

* **onUnstable** – Notifies when code has entered and is executing within the Angular zone.
* **onMicrotaskEmpty** - Notifies when no more microtasks are queued for execution. Angular subscribes to this internally to signal that it should run change detection.
* **onStable** – Notifies when the last **onMicroTaskEmpty** has run, implying that all tasks have completed and change detection has occurred.
* **onError** – Notifies when an error has occurred. Angular subscribes to this internally to send uncaught errors to its own error handler, i.e. the errors you see in your console prefixed with 'EXCEPTION:'.

We can inject the **NgZone** service in our component/services/etc. and can subscribe to these observables.

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

@Injectable()

export class OurZoneWatchingService() {

constructor(private ngZone: NgZone) {

this.ngZone.onStable.subscribe(this.onZoneStable);

}

onZoneStable() {

console.log('We are stable');

}

}

Subscribing to these can help you determine if your code is unexpectedly triggering change detection as a result of operations that do not affect application state.

**45. What are <code>@HostListener</code> and <code>@HostBinding</code>? How can detect events in or set properties on the parent element of a directive?**

**@HostListener** is a function decorator that accepts an event name as the first argument and the parameters array as the second argument. When that event gets fired on the host element it calls the associated function.

@Directive({

selector: `[appConfirm]`

})

export class ConfirmDirective {

**@HostListener('click', ['$event'])**

confirmFirst(event: Event) {

return window.confirm('Are you sure you want to do this?');

}

}

**@HostBinding** let us set properties on the host element or the component that hosts the directive.

@Directive({

selector: `[appConfirm]`

})

export class ConfirmDirective {

@HostBinding('style.color') color: string;

**@HostListener('click', ['$event'])**

confirmFirst(event: Event) {

this.color = 'hotpink';

}

}

**46. What are <code>TemplateRef</code> and <code>ViewContainerRef</code>?**

* **TemplateRef** – As the name suggest the **TemplateRef** is just a reference to the template.
* **ViewContainerRef** – Templates contain DOM elements, so if we need to insert an element, we need a place to “put” it. In Angular, this place is called a **container**. **ViewContainerRef** is a reference to the container.

The **ViewContainerRef** also exposes the **createEmbeddedView()** method that takes a template and renders its content as a **sibling** to the container. The following code shows a custom directive for adding or removing the template based on the condition just like **\*ngIf**.

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

export class MyNgIfDirective {

constructor(

private templateRef: TemplateRef,

private viewContainer: ViewContainerRef) { }

@Input() set myNgIf(condition: boolean) {

if (condition) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear();

}

}

}

**47. Explain AOT in Angular?**

Every Angular application requires a compilation process before they can run. The initial compiler is called JiT (Just-in-Time) compiler. As for AoT, it stands for the Ahead-of-Time compiler. Compared to the JiT compilation performed by Angular at run-time, AoT provides a smaller bundle with faster rendering in the browser.

The gist of AoT is moving the compilation from run-time to the building process. That means, first we can remove the JiT compiler from the bundle to have a smaller build, and second, the browser can execute the code without waiting for JiT in the run-time which leads to a faster rendering speed. Besides, AoT compiler also inlines HTML templates and CSS files and help reduce the amount of asynchronous requests sent by the application.

However, AoT is not perfect. The main limitation is that AoT, due to the way it compiles the raw code, cannot be used with common code patterns, for example, default exports from modules, template literals for templates, and functions in providers, routes, or declarations. Another problem with AoT is that when the application reaches certain complexity, the AoT bundle compared to JiT bundle can actually takes up more space.

We can generate a build in AoT by using the Angular Cli comment **ng build --aot** or **ng build --prod**.

**48. What is Traceur compiler?**

Traceur compiler is a Google project. It compiles ECMAScript Edition 6 (ES6) (including classes, generators and so on) code on the fly to regular Javascript (ECMAScript Edition 5 [ES5]) to make it compatible for the browser.

Traceur itself is written in ES6, compiled to ES5.

**49. What is the difference between <code>constructor</code> and <code>ngOnInit</code>?**

The **Constructor** is a default method of the class that is executed when the class is instantiated (by calling with **new**) and ensures proper initialization of fields in the class and its subclasses. It is a Typescript feature and it is used only for a class instantiations and nothing to do with Angular 4. Angular analyse the paramters of constructor to inject any dependencies.

**ngOnInit** is a life cycle hook called by Angular 4 to indicate that Angular is done creating the component. It is called after the **constructor**.

Mostly we use **ngOnInit** for all the initialization/declaration. The constructor should only be used to initialize class members and inject dependencies.

**50. What is <code>:host-context</code> pseudo-class selector?**

Sometimes, we will want to have a component apply a style to some element outside of it. This does not happen often, but one possible common use case is for theme enabling classes.

For example, let's say that we would like to ship a component with multiple alternative themes. Each theme can be enabled via adding a CSS class to a parent element of the component.

Here is how we could implement this use case using the **:host-context** selector:

@Component({

selector: 'themeable-button',

template: `

Themeable Button

`,

styles: [`

:host-context(.red-theme) .btn-theme {

background: red;

}

:host-context(.blue-theme) .btn-theme {

background: blue;

}

`]

})

export class ThemeableButtonComponent {

}

These themed styles are deactivated by default. In order to activate one theme, we need to add to any parent element of this component one of the theme-activating classes. For example,

<div class="blue-theme">

<themeable-button></themeable-button>

</div>

**51. How Two-way data binding works in Angular 2? Is it same like Anglar 1.x? Explain the Two-way data binding in Angular?**

Angular 2.x or higher versions doesn't come with such a (built-in) two-way data binding anymore. However, there's one directive in Angular that implements two-way data binding: **ngModel**. But it does not work like Angular 1.x. In fact, it turns out that two-way data binding really just boils down to event binding and property binding.

Consider the below snippet:

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

<p>Hello {{username}}!</p>

As said earlier, this uses property and event binding to bind the data between model and view. Without the shorthand syntax, this same code will look like:

<input [ngModel]="username" (ngModelChange)="username = $event">

<p>Hello {{username}}!</p>

The property binding **[ngModel]** takes care of updating the underlying input DOM element. The event binding **(ngModelChange)** notifies the outside world when there was a change in the DOM. **$event** is the payload of the emitted event. In other words, **ngModelChange** takes care of extracting **target.value** from the inner **$event** payload, and simply emits that.

Using this knowledge, we can build our own custom two-way data bindings with the use of property bindings and event emitters.

**52. What is the difference between Template driven forms and Model driven forms (or Reactive forms)?**

**Template driven form** is something when we write our logic, validation, binding, controls and everything in our templates which is html. The pros and cons: Easier to use, Suitable for simple scenarios, Uses Two way data binding (using **[(NgModel)]** syntax), Minimal component code, Automatic track of the form and its data (handled by Angular), Unit testing is hard

**Model driven form or Reactive form** holds all the logic and validation in the controller or component. It is a good default choice if we need larger forms and lots of validations and logic. The pros and cons: Easier Unit testing, More flexible, so Handles any complex scenarios., Reactive transformations can be made possible such as Adding elements dynamically, No data binding is done, More component code and less HTML markup.

**53. Explain the Template Driven Forms.**

Consider the below HTML template for a simple form which was built according to the template driven way. Note, inorder to use **ngForm**, we must add **FormsModule** in the **imports** array of **ngModule**.

<section class="sample-app-content">

<h1>Template-driven Form Example:</h1>

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

<p>

<label>Username:</label>

<input type="text" [(ngModel)]="user.username" required>

</p>

<p>

<label>Password:</label>

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

</p>

<p>

<button type="submit" [disabled]="!form.valid">Submit</button>

</p>

</form>

</section>

Once we import **FormsModule**, **NgForm** directive becomes active by default on all **<form>** tags. We can export **NgForm** into a local template variable in this case **form**. To import it we need to use **NgForm** as **ngForm**. We have two controls in this form: username and password both are required fields (marked with the **required** attribute).

The submit button is only enabled if both required fields are filled in. Also, the form will trigger the controller method **onSubmit** on submission. Angular is actually tracking the validity state of the whole form as well, using it to enable/disable the submit button. This functionality is actually common to both template-driven and model-driven forms. And css state classes are added to the elements based on the state. Since this is a template-driven form, there won't be much logic in the controller. Following is the component code for this template.

@Component({

selector: "template-driven-form",

templateUrl: 'template-driven-form.html'

})

export class TemplateDrivenForm {

user: Object = {};

onSubmitTemplateBased() {

console.log("Form is submitted.");

}

}

**54. Explain Model Driven Forms or Reactive Forms.**

A model driven form looks on the surface pretty much like a template driven form. But in order to be able to create this type of forms, we need first to import a different module into our application:

import {ReactiveFormsModule} from "@angular/forms";

@NgModule({

declarations: [App],

imports: [BrowserModule, ReactiveFormsModule],

bootstrap: [App]

})

export class AppModule {}

Instead of **FormsModule**, we imported **ReactiveFormsModule**. This will load the reactive forms directives instead of the template driven directives. The template would be look like this:

<section class="sample-app-content">

<h1>Model-driven Form Example:</h1>

<form [formGroup]="form" (ngSubmit)="onSubmit()">

<p>

<label>Username:</label>

<input type="text" formControlName="username">

</p>

<p>

<label>Password:</label>

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

</p>

<p>

<button type="submit" [disabled]="!form.valid">Submit</button>

</p>

</form>

</section>

There is a **formGroup** directive applied to the whole form, binding it to a controller variable named **form**. The **required** validator is removed and the logic is moved to the controller. Now the controller:

import { FormGroup, FormControl, Validators, FormBuilder } from '@angular/forms';

@Component({

selector: "model-driven-form",

templateUrl: 'model-driven-form.html'

})

export class ModelDrivenForm {

form: FormGroup;

firstName = new FormControl("Test User", Validators.required);

constructor(fb: FormBuilder) {

this.form = fb.group({

"firstName": this.firstName,

"password": ["", Validators.required]

});

}

onSubmitModelBased() {

console.log("model-based form submitted");

console.log(this.form);

}

}

We can see that the form is really just a **FormControl**, which keeps track of the global validity state. The controls themselves can either be instantiated individually or defined using a simplified array notation using the form builder. In the array notation, the first element of the array is the initial value of the control, and the remaining elements are the control's validators. In this case both controls are made mandatory via the Validators.required built-in validator.

**55. What's the difference between <code>NgForm</code>, <code>FormGroup</code>, and <code>FormControl</code>?**

The **ngForm** is used in Template-driven form where Angular activate for all the forms by default when we use **FormsModule**. This type of form is very well-suited to be used with ng-model, and ideal for migrating Angular 1 forms to Angular 4.

The Reactive forms or Model-driven forms uses **formGroup** directive applied to the whole form.

In Reactive forms or Model-driven forms, each form elements are controlled by a **FormControl**, which keep the elements states. The controls themselves can either be instantiated individually or defined using a simplified array notation using the form builder. A **FormControl** requires two properties: an initial value and a list of validators.

**56. How do you add form validation to a form built with <code>FormBuilder</code>?**

The **FormControl** requires two properties: an initial value and a list of validators. Angular provides many validators out of the box. They can be imported along with the rest of dependencies for procedural forms.

From the previous login form, we can quickly and easily add validation.

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

import { Validators, FormBuilder, FormControl } from '@angular/forms';

@Component({

*// ...*

})

export class AppComponent {

username = new FormControl('', [

Validators.required,

Validators.minLength(5)

]);

password = new FormControl('', [

Validators.required

]);

loginForm: FormGroup = this.builder.group({

username: this.username,

password: this.password

});

constructor(private builder: FormBuilder) { }

}

First, we need to import **Validators** along with the other form modules. In this example we have used Angular built in validators.

<form [formGroup]="loginForm" (ngSubmit)="login()">

<div>

<label for="username">username</label>

<input type="text" name="username" id="username" [formControl]="username">

<div [hidden]="username.valid || username.untouched">

<div>

The following problems have been found with the username:

</div>

<div [hidden]="!username.hasError('minlength')">

Username can not be shorter than 5 characters.

</div>

<div [hidden]="!username.hasError('required')">

Username is required.

</div>

</div>

</div>

<div>

<label for="password">password</label>

<input type="password" name="password" id="password" [formControl]="password">

<div [hidden]="password.valid || password.untouched">

<div>

The following problems have been found with the password:

</div>

<div [hidden]="!password.hasError('required')">

The password is required.

</div>

</div>

</div>

<button type="submit" [disabled]="!loginForm.valid">Log In</button>

</form>

We are using **.valid** and **.untouched** to determine if we need to show errors - while the field is required, there is no reason to tell the user that the value is wrong if the field hasn't been visited yet.

For built-in validation, we are calling **.hasError()** on the form element, and we are passing a string which represents the validator function we included. The error message only displays if this test returns true.

**57. How do you add custom validators in Angular?**

Angular allows us to add our own custom validators with the built-in validators easily. All we need to do is write our validator function that takes the **FormControl** instance. Let us consider, we need to verify the password from the above form should contains a exclamation mark. Here is how we can do it.

function hasExclamationMark(input: FormControl) {

const hasExclamation = input.value.indexOf('!') >= 0;

return hasExclamation ? null : { needsExclamation: true };

}

password = new FormControl('', [

Validators.required,

hasExclamationMark

]);

A simple function takes the **FormControl** instance and returns **null** if everything is fine. If the test fails, it returns an object with an arbitrarily named property. The property name is what will be used for the **.hasError()** test. The HTML would be:

*<!-- --->*

<div [hidden]="!password.hasError('needsExclamation')">

Your password must have an exclamation mark!

</div>

*<!-- --->*

**58. What is async validation and how is it done?**

Async Validators are used propably when we want to done our validators asynchronously. For example, checking the back end server. In our form example, if we want to check the username that entered is available, we can do that using async validators.

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

import { UsernameService } from '../username.service';

export class ValidateUsernameNotTaken {

static createValidator(usernameService: UsernameService) {

return (control: AbstractControl) => {

return usernameService.checkUsernameTaken(control.value).map(res => {

return res ? null : { usernameTaken: true };

});

};

}

}

We can then add our custom validators in the validators array.

username = new FormControl('', [

Validators.required,

Validators.minLength(5)

, ValidateUsernameNotTaken.createValidator(this.UsernameService)

]);

In HTML, we could check the errors as follows:

<div \*ngIf="myForm.get('email').errors && myForm.get('email').errors.emailTaken">

Oh noes, this email is already taken!

</div>

We could also display an error message if the async validation errors-out with our **usernameTaken** error.

**59. How would you select all the child components' elements?**

We can use **ElementRef** with **querySelector** (**elRef.nativeElement.querySelector(selector);**). But a declarative way can be used instead to access elements in the view directly:

<input #myElm>

@ViewChild('myElm') input;

ngAfterViewInit() {

console.log(this.input.nativeElement.value);

}

**60. What is Observables in Angular?**

**Observable** is one of the exciting features of Angular. This isn't an Angular specific feature, but rather a proposed standard for managing async data that will be included in the release of ES7. Observables open up a continuous channel of communication in which multiple values of data can be emitted over time. From this we get a pattern of dealing with data by using array-like operations to parse, modify and maintain data. Angular uses observables extensively - we'll see them in the HTTP service and the event system.

To use observables, Angular uses a third-party library called Reactive Extensions (RxJS).

**61. What is the difference between an observable and a promise?**

**Promise** - A **Promise** handles a single event when an async operation completes or fails. Promise helps us run our functions asynchronously, and use their return values (or exceptions) but only once when executed.

**Observable** - An **Observable** is like a Stream (in many languages) and allows us to pass zero or more events where the callback is called for each event.

**Observable** also has the advantage over **Promise** to be cancelable. If the result of an HTTP request to a server or some other expensive async operation isn't needed anymore, the **Subscription** of an **Observable** allows to cancel the subscription, while a **Promise** will eventually call the success or failed callback even when you don't need the notification or the result it provides anymore. (Note: Some **Promise** libraries support cancellation.)

**62. What is the difference between an observable and a subject?**

An observable, by definition is a data producer. It can produce data over time.

On the other hand, a subject can act as both: a data producer and a data consumer. Thus, a subject can be subscribed to, like an observable and a subject can subscribe to other observables.

So we can conclude that the critical difference between a subject and an observable is: when a subject produces data, all of its subscribers will receive the same data. This behavior is different from observables, where each subscription causes an independent execution of the observable.

Probably a more important distinction between Subject and Observable is that a Subject has state, it keeps a list of observers. On the other hand, an Observable is really just a function that sets up observation. A subject can be used as an observer to subscribe to any observable.

**63. How to implement multiple API calls using RxJS?**

We can use one of the RxJS operators **MergeMap** also known as **flatMap** to combine more than one observables. This creates a nested **Observable** in a **Observable**. The **mergeMap** operator helps us by subscribing and pulling the value out of the inner Observable and passing it back to the parent stream. This condenses our code quite a bit and removes the need of a nested subscription.

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

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

import { Observable } from 'rxjs/Observable';

import { mergeMap } from 'rxjs/operators';

@Component({

selector: 'app-root',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

homeworld: Observable<{}>;

constructor(private http: HttpClient) { }

ngOnInit() {

this.homeworld = this.http.get('/api/product/1').pipe(

mergeMap(product => this.http.get(product.description))

);

}

}

An another operator called **forkJoin** allows us take a list of Observables and execute them in parallel. This is same like **Promise.all()**. Once every **Observable** in the list emits a value the **forkJoin** will emit a single Observable value containing a list of all the resolved values from the **Observables** in the list.

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

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

import { Observable } from 'rxjs/Observable';

import { forkJoin } from 'rxjs/observable/forkJoin';

@Component({

selector: 'app-root',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

loadedCharacter: {};

constructor(private http: HttpClient) { }

ngOnInit() {

let api1 = this.http.get('API1\_ENDPOINT');

let api2 = this.http.get('API2\_ENDPOINT');

forkJoin([api1, api2]).subscribe(results => {

*// results[0] is our api1*

*// results[1] is our api2*

});

}

}

**64. How would you cache an observable data?**

Caching is important for performance, especially on bandwidth restricted devices and slow networks. We can easily add caching to the observable by adding **publishReplay(1)** and **refCount**.

getData(){

if(!this.\_cachedDate){

this.\_cachedDate = this.\_http.get('API\_ENDPOINT')

.map((res:Response) => res.json())

.publishReplay(1)

.refCount();

}

return this.\_cachedDate;

}

**publishReplay(1)** tells RxJS to cache the most recent value which is perfect for single value http calls. **refCount()** is used to keep the observable alive for as long as there are subscribers.

Cacheing HTTP results will be useful when we need to get some data in multiple routes but, once we get it, we can reuse it in the routes that needs it, therefor no need to make another call to backend.

**65. How do you improve performance of an Angular 4 application?**

* **Using OnPush** - By default, Angular runs change detection on all components every time something changes in your app. We can set the **ChangeDetectionStrategy** of our component to **ChangeDetectionStrategy.OnPush**. This tells Angular that the component only depends on his **Inputs** and needs to be checked in only the following cases: The **Input** reference changes, An event occurred from the component or one of his children and You run change detection explicitly.
* **Using TrackBy** - If we have a collection of data, Angular can't keep track of items in the collection and has no knowledge of which items have been removed or added. As a result, Angular needs to remove all the DOM elements associated with the data and create them again. This can mean a lot of DOM manipulations. If we provide a trackBy function, Angular can track which items have been added or removed according to the unique identifier and only create or destroy the things that have changed.
* **Avoid computing values in templates** - If we bind a function in our template, Angular needs to re-run your function in every change detection cycle. And if the function performs expensive tasks, it can be costly.
* **Disable Change Detection** - Updating the user interface whenever new data arrives can be expensive. A more efficient way would be to check and update the user interface every X seconds. We can do that by detaching the component's change detector and conducting a local check every x seconds.
* **Using Lazy Loading** - By default, Webpack will output all your app's code into one large bundle. Lazy loading gives you the ability to optimize your application load time by splitting the application to feature modules and loading them on-demand.

**66. What is <code>providedIn</code> used for?**

Angular 6 introduced new way to register a service into Angular Dependency. This is also called **Tree Shakable providers** and we can use to by using **providedIn** property in the **@Injectable**.

We can **providedIn** an injectable to **root** or any other module. In fact **root** is an alias to inject in **AppModule**.

The main benefit of this new syntax is that the services will be bundled and included only if they are really used.

@Injectable({

providedIn: 'root'

})

export class SampleService {

constructor() {

}

}

**67. What is <code>useClass</code> and <code>useValue</code>?**

The **useClass** in the provider key let us create and return a new instance of the specified class. We can use this if we are asking for this service anywhere in our application we want to use a different service instead. That is we can substitute an alternative implementation for a common or default class.

export class AnotherService {

sayHello() {

console.log(`From AnotherService`);

}

}

@Injectable({

providedIn: 'root',

useClass: AnotherService,

})

export class SampleService {

sayHello() {

console.log(`From SampleService`);

}

}

Now if we are inject the **SampleService** and invoke **sayHello** it will log **From AnotherService**.

The **useValue** is used to provide a single value. This way we can pass a single value and inject them into components. This technique can be used to provide runtime configuration constants such as website base addresses and feature flags.

{ provide: API\_KEY, useValue: 'API\_KEY' },

**68. What is <code>useExisting</code>?**

The **useExisting** provider key let us map one token to another. In effect, the first token is an alias for the service associated with the second token, creating two ways to access the same service object.

{ provide: AnotherLogger, useExisting: LoggerService },

Image we have **LoggerService**, but we want to update with **AnotherLogger**. But we can not do for any reason. **useExisting** let us use do this with an alias. Whenever we inject **AnotherLogger** we will get **LoggerService**.

{ provide: API\_KEY, useValue: 'API\_KEY' },

**69. What is <code>useFactory</code>?**

We can use **useFactory** at runtime to decide which kind of service we want to return if it got requested by another service in our application.

export class AnotherService {

printHello() {

console.log('Hello from AnotherService');

}

}

export function ServiceFactory() {

return new AnotherService();

}

@Injectable({

providedIn: 'root',

useFactory: ServiceFactory

})

export class SampleService {

printHello() {

console.log('Hello from SampleService');

}

}

The injector provides the dependency by invoking the factory function.

**70. What is ngUpgrage?**

NgUpgrade is a library put together by the Angular team, which we can use in our applications to mix and match AngularJS and Angular components and bridge the AngularJS and Angular dependency injection systems.

One of the primary tools provided by ngUpgrade is called the UpgradeModule. This is a module that contains utilities for bootstrapping and managing hybrid applications that support both Angular and AngularJS code.

What happens on top is that components and services managed by one framework can interoperate with those from the other framework. This happens in three main areas: Dependency injection, the DOM, and change detection.