**1. AOT VS JIT:**

In Angular we have 2 models of compilation

JIT - Just-in-Time Compilation : **JIT compilation as the name implies, compiles the application Just-in-Time in the browser at runtime.**

AOT - Ahead-of-Time Compilation : **AOT compilation compiles the application at build time.**

1. By default, with the development build we get JIT compilation. This is how it works. The application code along with the angular compiler is downloaded by the browser. At runtime, when a request is issued to the application, the JIT-compiler in the browser compiles the application code before it is executed. This means our user who made that first request has to wait for the application to compile first.

2. The vendor bundle contains the compiler along with the angular framework. The compiler code is roughly half of the Angular framework.

3. There is a tool called source-map-explorer that we can use to inspect the JavaScript bundles. This tool analyzes the source map generated with the bundle and draws a map of all dependencies.

4. To be able to use this tool we have to install it first. To install this tool, execute the following command

npm install source-map-explorer --save-dev

5. Once we have the tool installed, if you have not done the development build yet, do the developments build using the following command.

ng build

6. Once the build is complete, you will have the JavaScript bundles along with the source map files. Now execute the following command.

node\_modules\.bin\source-map-explorer dist\vendor.bundle.js

7. The above command runs the source-map-explorer against the vendor bundle and we see the graph of it. Notice the angular compiler is around 45% percent of the bundle size. As this is development build and not optimised, notice the total size of the bundle is 2.19 MB.

8. **With AOT compilation the angular application is pre-compiled. So this means the browser loads executable code so it can render the application immediately, without waiting to compile the application first.**

9. This also mean with AOT, as the application is already pre-compiled, there is also no need for the browser to download the Angular compiler. As we already know, the compiler code is roughly half of the Angular framework, so omitting it dramatically reduces the application size.

10. By default, the production build is Ahead-of-Time compiled. So there is no need to bundle up the angular compiler code in the vendor bundle. This brings down the vendor bundle size by almost 50%. In addition it is also minified, uglified and tree-shaked to remove any code that we are not referencing in our application. So the bundler size is further reduced.

11. Now, execute the following command to generate a production build. Notice I have also turned on sourcemap option. Without the sourcemap we will not be able to use the source-map-explorer tool.

ng build --prod --sourcemap true

12. Once the production build is complete, execute the following command. Vendor bundle name in your production build may be slightly different. Change it accordingly and execute the command.

node\_modules\.bin\source-map-explorer dist\vendor.7e385ef294695236ffd1.bundle.js

13. The AOT compiler also detects and reports template binding errors at build time itself. Let us understand this with an example.

Include the following function HomeComponent class in home.component.ts file

getText(): string {

return 'Hello Pragim';

}

15. In home.component.html include the following [div] element. Notice I have deliberately mis-spelled the getText() function name.

[div [innerText]='getTex()']

16. Save changes, and execute the following command. This command does a development build in-memory. At the moment we are not using AOT, so we will not know about the template binding error that is introduced above. Notice at build time we do not see any errors.

ng serve

**Q). Promises (vs) Observables:**

To work with asynchronous data we can use either Promises or Observables

1. A Promise emits a single value where as an Observable emits multiple values over a period of time.

You can think of an Observable like a stream which emits multiple items over a period of time and the same callback function is called for each item emitted. So with an Observable we can use the same API to handle asynchronous data whether that data is emitted as a single value or multiple values over a period of time.

2. A Promise is not lazy where as an Observable is Lazy.

3. A Promise cannot be cancelled where as an Observable can be cancelled using the unsubscribe() method

4. Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc.

**3.** **ng serve vs ng build:**

A)

1. To see the list of all options that we can use with "ng serve" command use --help option

ng serve --help

2. The following command, builds and launches the application in your default browser.

ng serve --open

3. The following table shows the common options, alias, default value and their purpose

Option Alias Default Purpose

--watch -w true Run build when files change

--live-reload -lr true Whether to reload the page on change

--open -o false Opens the url in default browser

--port -p 4200 The port on which the server is listening

--extract-css -ec Extract css from global styles onto css files instead of js ones.

**ng serve vs ng build**

**ng serve**

1. Compiles and serves the application from memory

2. Does not write the build files to the disk

3. Typically used to run the application on local development machine

4. Cannot be used for deploying the build to another server (Ex. Testing, Staging or Production server)

**ng build**

1. Compiles the application to the "dist" folder

2. Can be used to produce both development & production builds

3. Typically used to deploy the application on another server

**5. Angular life cycle hooks:**

A) A component has a lifecycle managed by Angular. Angular

1. Creates the component

2. Renders the component

3. Creates and renders the component children

4. Checks when the component data-bound properties change, and

5. Destroys the component before removing it from the DOM

To tap into and react when these life cycle events occur, angular offers several lifecycle hooks

The 3 most commonly used hooks are

**ngOnChanges** - Executes, every time the value of an input property changes. The hook method receives a SimpleChanges object containing current and previous property values. This is called before ngOnInit.

**ngOnInit** - Executes after the constructor and after ngOnChange hook for the first time. It is most commonly used for component initialisation and retrieving data from a database.

**ngDoCheck()-**Detect and act upon changes that Angular can't or won't detect on its own.

Called during every change detection run, immediately after ngOnChanges() and ngOnInit().

**ngOnDestroy** - Executes just before angular destroys the component and generally used for performing cleanup.

There are 3 simple steps to use the Life Cycle Hooks

Step 1 : Import the Life Cycle Hook interface. For example, to use ngOnInit() life cycle hook, import OnInit interface.

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

Step 2 : Make the component class implement the Life Cycle Hook interface, using the implements keyword as shown below. This step is optional, but good to have so you will get editor support and flags errors at compile time if you incorrectly implement the interface method or make any typographical errors.

export class SimpleComponent implements OnInit { }

Step 3 : Write the implementation code for the life cycle interface method. Each interface has a single hook method whose name is the interface name prefixed with ng.

ngOnInit() {

console.log('OnInit Life Cycle Hook');

}

Let's understand ngOnChanges life cycle hook with a simple example. Here is what we want to do. As soon as the user starts typing into the text box, we want to capture the current and previous value and log it to the browser console. We can very easily achieve this by using the ngOnChanges life cycle hook.

ngOnChanges, is called every time the value of an input property of a component changes. So first let's create a SimpleComponent with an input property as shown below. We will continue with the example we worked with in our previous video. Add a new folder in the App folder and name it Others. Add a new TypeScript file to this folder and name it - simple.component.ts. Copy and paste the following code which is commented and self explanatory.

// Step 1 : Import OnChanges and SimpleChanges

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

// The selector "simple" will be used as the directive

// where we want to use this component. Notice we are

// also using the simpleInput property with interpolation

// to display the value it receives from the parent

// component

@Component({

selector: 'simple',

template: `You entered : {{simpleInput}}`

})

// Step 2 : Implement OnChanges Life Cycle Hook interface

export class SimpleComponent implements OnChanges {

// Input property. As and when this property changes

// ngOnChanges life cycle hook method is called

@Input() simpleInput: string;

// Step 3 : Implementation for the hook method

// This code logs the current and previous value

// to the console.

ngOnChanges(changes: SimpleChanges) {

for (let propertyName in changes) {

let change = changes[propertyName];

let current = JSON.stringify(change.currentValue);

let previous = JSON.stringify(change.previousValue);

console.log(propertyName + ': currentValue = '

+ current + ', previousValue = ' + previous);

// The above line can be rewritten using

// placeholder syntax as shown below

// console.log(`${propertyName}: currentValue

// = ${current }, previousValue = ${previous }`);

}

}

}

**5. Why do we need a service in Angular**

1.A service in Angular is generally used when you need to reuse data or logic across multiple components. Anytime you see logic or data-access duplicated across multiple components, think about refactoring that piece of logic or data-access code into a service. Using a service ensures we are not violating one of the Software principles - DRY ((Don't repeat yourself). The logic or data access is implemented once in a service, and the service can be used across all the components in our application.

2.Without the service you would have to repeat your code in each component. Imagine the overhead in terms of time and effort required to develop, debug, test and maintain the duplicated code across multiple places instead of having that duplicated code at one central place like a service and reusing that service where required.

**6. Difference between constructor() and ngOnInit()**

1.A class constructor is automatically called when an instance of the class is created. It is generally used to initialise the fields of the class and it's sub classes.

2.ngOnInit is a life cycle hook method provided by Angular. ngOnInit is called after the constructor and is generally used to perform tasks related to Angular bindings. For example, ngOnInit is the right place to call a service method to fetch data from a remote server. We can also do the same using a class constructor, but the general rule of thumb is, tasks that are time consuming should use ngOnInit instead of the constructor. As fetching data from a remote server is time consuming, the better place for calling the service method is ngOnInit.

**the dependency injection** is done using the class constructor and the actual service method call is issued from ngOnInit life cycle hook as shown below

**7. What is Dependency Injection? How dependency injection works in angular?**

Let us understand Dependency Injection in Angular with an example. Consider this piece of code in EmployeeListComponent.

export class EmployeeListComponent implements OnInit {

private \_employeeService: EmployeeService;

constructor(\_employeeService: EmployeeService) {

this.\_employeeService = \_employeeService;

}

ngOnInit() {

this.\_employeeService.getEmployees()

.subscribe(

employeesData =] this.employees = employeesData,

error =] this.statusMessage = 'Error');

}

// Rest of the code

}

1. We register a service with the angular injector by using the providers property of @Component decorator or @NgModule decorator.

2. When a component in Angular needs a service instance, it does not explicitly create it. Instead it just specifies it has a dependency on a service and needs an instance of it by including the service as a constructor parameter.

3. When an instance of the component is created, the angular injector creates an instance of the service class and provides it to component constructor.

4. So the component which is dependent on a service instance, receives the instance from an external source rather than creating it itself. This is called Dependency Injection.

**What is Dependency Injection:**

It's a coding pattern in which a class receives its dependencies from an external source rather than creating them itself.

So if we relate this definition to our example, EmployeeListComponent has a dependency on EmployeeService. The EmployeeListComponent receives the dependency instance (i.e EmployeeService instance) from the the external source (i.e the angular injector) rather than creating the instance itself.

8. **Singleton service**

1.In our previous question we discussed why should we use dependency injection and the benefits it provides. One of the benefits of dependency injection is that it allows us to share data and functionality easily as the angular injector provides a Singleton i.e a single instance of the service.

2.So in this question let us see how we can use Angular services and dependency injection to create a Singleton i.e a single instance of the service which enables us to share data and functionality across multiple components in our application.

**9. Different ways of passing data between components.**

If the components are nested, then there is a parent child relationship between those components.

1. To pass data from the parent to child component we use input properties.

2. To pass data from the child component to parent component we can either use output properties or template reference variables.

[**https://www.youtube.com/watch?v=UH59HWuj\_tY**](https://www.youtube.com/watch?v=UH59HWuj_tY)

**10. What is Router Outlet?**

The Router-Link, RouterLink-Active and Router-Outlet are directives provided by the Angular RouterModule package. It is Provides the navigation and URLs manipulation capabilities.

“A router outlet will emit an activate event any time a new component is being instantiated, and a deactivate event when it is being destroyed.”

**Router-outlet directive:** - Router-outlet directive is used to render the components for specific location of your applications.

Both the template and templateUrl render the components where you use this directive.  
**Syntax -**

<router-outlet> </router-outlet>

Example as,

<div **class**='container'>

<div **class**='row'>

<router-outlet></router-outlet>

</div>

</div>

# 11. Is it possible to have a multiple router-outlet in the same template?

Yes! We can use multiple router-outlets in same template by configuring our routers and simply add the router-outlet name. You can see in the example.

**Syntax-**

<div class="row">

<div class="user">

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

</div>

<div class="userInfo">

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

</div>

</div>

And setup your route config:

**const** routes: **Routes** = [

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

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

{ path: 'user', component: **userComponent**, children: [

{ path: 'userList', component: **userListComponent**, outlet: 'userList' },

{ path: ':id', component: **userInfoComponent**, outlet: 'userInfo' }]

}];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule],

providers: []

})

**export** **class** RoutingModule { }

**12. Interfaces**

What is an Interface in TypeScript

If you have experience with any object oriented oriented programming language like C#, Java or C++, then you know an interface is an abstract type. It only contain declarations of properties, methods and events. The implementation for the interface members is provided by a class that implements the interface.

If a class that implements the interface fails to provide implementation for all the interface members, the language compiler raises an error alerting the developer that something has been missed.

In general, an interface defines a contract (i.e the shape or structure of an API), that an implementing class must adhere to. Even in TypeScript we use the interface for the same purpose.

We know TypeScript is a strongly typed language. This means every property we define in a TypeScript class has a type associated with it. Similarly every method parameter and return type has a type. As you know by now, TypeScript has several built-in pre-defined types like string, number, boolean etc.

However, the business objects that we usually create in real-world applications like Employee, Customer, Order, Invoice etc, does not have a pre-defined type. In this case, we can use an interface to create a custom type for our business object.

Let us understand this with an example. We will use the same example we worked with in Part 22 of Angular 2 tutorial.

Notice the below line of code in EmployeeListComponent class. The property "employees" is defined as an array of any type.

employees: any[];

Since we do not have a Type for employee object, we specified the type as any.

There are 2 problems with the above line of code

1. For the object properties in the array we do not get intellisense

2. Since we do not get intellisense, we are prone to making typographical errors and the compiler will not be able to flag them as errors. We will come to know about these errors only at runtime.

Let's create a Type for employee using an interface as shown below. Add a new TypeScript file to the employee folder. Name it employee.ts. Copy and paste the following code.

export interface IEmployee {

code: string;

name: string;

gender: string;

annualSalary: number;

dateOfBirth: string;

}

With this IEmployee interface in place, we can now import and use the interface type as the type for "employees" property. The code in EmployeeListComponent class is shown below.

import { IEmployee } from './employee';

export class EmployeeListComponent {

employees: IEmployee[];

}

Since we have specified a type for the "employees" property, we now get intellisense for the object properties in the array

As you can see from the image below, if we make any typographical errors with the property names, we will get to know these errors right away at compile time.

Interfaces in TypeScript

1. Use interface keyword to create an interface

2. It is common to prefix the interface name with capital letter I. However, some interfaces in Angular does not have the prefix I. For example, OnInit interface

3. Interface members are public by default and does not require explicit access modifiers. It is a compile time error to include an explicit access modifier. You will see an error message like - public modifier cannot appear on a type member.

4. A class that implements an interface must provide implementation for all the interface members unless the members are marked as optional using the ? operator

5. Use the implements keyword to make a class implement an interface

6. TypeScript interfaces exist for developer convenience and are not used by Angular at runtime. During transpilation, no JavaScript code is generated for an interface. It is only used by Typescript for type checking during development.

7. To reduce the amount of code you have to write, consider using short-hand syntax to initialise class properties with constructor parameters

**13. Lazy loading**

1. Application performance is an important part of web development. We always want our users to have the quickest and most preferment experience when they visit our apps.

2. Lazy loading is one of the features we can add to our app to improve performance. What exactly is lazy loading?

4. Lazy loading is the act of **only loading parts of our app when they are needed**.

**5. The benefits are:**

1. Keep the initial payload small

2. Smaller payloads lead to faster download speeds

3. Lower resource costs especially on mobile networks

4. If a user doesn't visit a section of our app, they won't ever download those resources

Eg:

const routes: Routes = [

{

path: 'dashboard',

loadChildren: 'app/dashboard/dashboard.module#DashboardModule',

}

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

<https://scotch.io/tutorials/lazy-loading-in-angular-v2>

**Q) Pipes:**

Pipes are used to transform data, when we only need that data transformed in a template.

**Pipes provided by Angular**

1. CurrencyPipe
2. DatePipe
3. DecimalPipe
4. JsonPipe
5. LowerCasePipe
6. UpperCasePipe
7. PercentPipe
8. SlicePipe
9. AsyncPipe

**AsyncPipe:**

With AsyncPipe we can use promises and observables directly in our template, without having to store the result on an intermediate property or variable.

AsyncPipe accepts as argument an observable or a promise, calls subcribe or attaches a then handler, then waits for the asynchronous result before passing it through to the caller.

**Custom pipe:**

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'filesize' })

export class FileSizePipe implements PipeTransform {

transform(size: number): string {

return (size / (1024 \* 1024)).toFixed(2) + 'MB';

}

}

**html:** {{ file.size | filesize }}

Q) **What is ViewEncapsulation?**

ViewEncapsulation decides whether the styles defined in a component can affect the entire application or not. There are three ways to do this in Angular:

1. Emulated: styles from other HTML spread to the component.

2. Native: styles from other HTML do not spread to the component.

3. None: styles defined in a component are visible to all components.

**Unsubscribe observable:**

Reactive-Extensions for JavaScript (or RxJS) introduces the concept of Observables to Angular. If you have been using version 1 of Angular then you are likely comfortable using Promises. And, while you might think that an Observable is just like a Promise you might be surprised (as I was) to learn that they are in fact very different.

First, Promises are eager and are executed immediately. Observables are not eager and are only executed when subscribed to.

Second, Promises are asynchronous. Observables can be either synchronous or asynchronous.

Third, Promises are expected to return only a single value (like a function). Observables can return zero, one or more (infinitely) values.

Let me get to the point. A subscription is created when we subscribe() to an observable. And it is important that we unsubscribe from any subscriptions that we create in our Angular components to avoid memory leaks.

ngOnDestroy(): Cleanup just before Angular destroys the directive/component. Unsubscribe observables and detach event handlers to avoid memory leaks.

1.import { ISubscription } from "rxjs/Subscription";

2.this.subscription = this.counter.subscribe(

(value) => this.count = value,

(error) => console.error(error),

() => console.log('complete')

);

3. ngOnDestroy() {

this.subscription.unsubscribe();

}

**ASYNC - WAIT:**

[**https://medium.com/@balramchavan/using-async-await-feature-in-angular-587dd56fdc77**](https://medium.com/@balramchavan/using-async-await-feature-in-angular-587dd56fdc77)