# Spread Operator

**The Spread Operator allows splitting an array to single arguments which are passed to the function as separate arguments**. The spread operator takes either an array or an object and expands it into its set of items. The **spread syntax** allows an iterable such as an array expression or string to be expanded in places where zero or more arguments or elements are expected, or an object expression to be expanded in areas where zero or more key-value pairs are expected.

 Example :

const array = [1, 2];

const combined = [...array, 3, 4];

console.log(combined);

 let obj = { ...object };

 myFunction(...iterableObject);

# Arrow functions

An **arrow function expression** is a syntactically compact alternative to a regular [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function#_blank)

Another significant advantage it offers is the fact that it does not bind its own **this**. In other words, the context inside arrow functions is lexically or statically defined.

The value of this inside arrow functions is not dependent on how they are invoked or how they are defined. It depends only on its enclosing context.

# Directives

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

## Types of directives in Angular

* Component
* Attribute Directive
* Structural Directive

The basic difference between a component and a directive is that a component has a template, whereas an attribute or structural directive does not have a template

# What Are @HostBinding() and @HostListener() in Angular?

**@HostListener()** function decorator allows you to handle events of the host element in the directive class.

**@HostBinding()** function decorator allows you to set the properties of the host element from the directive class.

# Renderer2

The **Renderer2** class is an abstraction provided by Angular in the form of a service that allows to manipulate elements of your app without having to touch the DOM directly.

Functions:

**addClass** : this.renderer.addClass(this.el.nativeElement, 'wild');

**removeClass** : this.renderer.removeClass(this.el.nativeElement, 'wild');

**createElement** : this.renderer.createElement('div');

**createText** : this.renderer.createText('Hello world!');

**appendChild** : this.renderer.appendChild(div, text);

**setAttribute** : this.renderer.setAttribute(this.el.nativeElement, 'aria-hidden', 'true');

**removeAttribute** : this.renderer.removeAttribute(this.el.nativeElement, 'aria-hidden');

**setStyle** : this.renderer.setStyle(this.el.nativeElement, 'border-left', '2px dashed olive');

**removeStyle** : this.renderer.removeStyle(this.el.nativeElement, 'border-left')

**setProperty** : this.renderer.setProperty(this.el.nativeElement, 'alt', 'Cute alligator');

# Content Projection:

Content projection allows you to insert a shadow DOM in your component. To put it simply, if you want to insert HTML elements or other components in a component, then you do that using the concept of content projection. In Angular, you achieve content projection using < ng-content >< /ng-content >.  You can make reusable components and scalable applications by properly using content projection.

When the data is produced by the Observable itself, we call it a **cold Observable**. When the data is produced outside the Observable, we call it a **hot Observable.**

**Observable                  ┃     BehaviorSubject/Subject**         ┃

┣━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╋━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┫

│ Is just a function, no state        │ Has state. Stores data in memory    │

├─────────────────────────────────────┼─────────────────────────────────────┤

│ Code run for each observer          │ Same code run                       │

│                                     │ only once for all observers         │

├─────────────────────────────────────┼─────────────────────────────────────┤

│ Creates only Observable             │Can create and also listen Observable│

│ ( data producer alone )             │ ( data producer and consumer )      │

├─────────────────────────────────────┼─────────────────────────────────────┤

│ Usage: Simple Observable with only  │ Usage:                              │

│ one Obeserver.                      │ \* Store data and modify frequently  │

│                                     │ \* Multiple observers listen to data │

│                                     │ \* Proxy between Observable  and     │

│                                     │   Observer

# tsconfig.json

The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

 This file is used to give the options about TypeScript used for the Angular JS project.

{

   "compilerOptions": {

      "target": "es5",

      "module": "commonjs",

      "moduleResolution": "node",

      "sourceMap": true,

      "emitDecoratorMetadata": true,

      "experimentalDecorators": true,

      "lib": [ "es2015", "dom" ],

      "noImplicitAny": true,

      "suppressImplicitAnyIndexErrors": true

   }

}

Following are some key points to note about the above code.

* The target for the compilation is es5 and that is because most browsers can only understand ES5 typescript.
* The sourceMap option is used to generate Map files, which are useful when debugging. Hence, during development it is good to keep this option as true.
* The "emitDecoratorMetadata": true and "experimentalDecorators": true is required for Angular JS decorators. If not in place, Angular JS application will not compile.

# package.json

This file contains information about Angular 2 project.

Some key points to note about the above code −

There are two types of dependencies, first is the dependencies and then there are dev dependencies. The dev ones are required during the development process and the others are needed to run the application.

The "build:watch": "tsc -p src/ -w" command is used to compile the typescript in the background by looking for changes in the typescript files.

# Observable.subscribe()

The observable subscribe method is used by angular components to subscribe to messages that are sent to an observable.

# Subject.next()

The subject next method is used to send messages to an observable which are then sent to all angular components that are subscribers (a.k.a. observers) of that observable.

<http://jasonwatmore.com/post/2018/06/25/angular-6-communicating-between-components-with-observable-subject>

<https://www.google.com/amp/s/scotch.io/tutorials/lazy-loading-in-angular-v2/amp>

# Lifecycle Hooks

Angular calls the lifecycle hook methods in the following sequence

1.      **ngOnChanges()** : Respond when Angular (re)sets data-bound input properties. The method receives a SimpleChanges object of current and previous property values.

Called before ngOnInit() and whenever one or more data-bound input properties change.

2.      **ngOnInit**() : Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.

Called once, after the first ngOnChanges().

3.      **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().

4.      **ngAfterContentInit**() : Respond after Angular projects external content into the component's view / the view that a directive is in.

Called once after the first ngDoCheck().

5.      **ngAfterContentChecked**() : Respond after Angular checks the content projected into the directive/component.

Called after the ngAfterContentInit() and every subsequent ngDoCheck().

6.      **ngAfterViewInit**() : Respond after Angular initializes the component's views and child views / the view that a directive is in.

Called once after the first ngAfterContentChecked().

7.      **ngAfterViewChecked**() : Respond after Angular checks the component's views and child views / the view that a directive is in.

Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked().

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

Called just before Angular destroys the directive/component.

# What are the key components of Angular?

Angular has the below key components,

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

# What is metadata?

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

* **Class decorators**, e.g. @Component and @NgModule
* **Property decorators** Used for properties inside classes, e.g. @Input and @Output
* **Method decorators** Used for methods inside classes, e.g. @HostListener
* **Parameter decorators** Used for parameters inside class constructors, e.g. @Inject

# What are the class decorators in Angular?

A class decorator is a decorator that appears immediately before a class definition, which declares the class to be of the given type, and provides metadata suitable to the type The following list of decorators comes under class decorators,

* @Component()
* @Directive()
* @Pipe()
* @Injectable()
* @NgModule()

# What are class field decorators?

The class field decorators are the statements declared immediately before a field in a class definition that defines the type of that field. Some of the examples are: @input and @output,

* @Input() myProperty;
* @Output() myEvent = new EventEmitter();

# What is angular CLI?

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

npm install @angular/cli@latest

# What is the difference between constructor and ngOnInit?

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

# What is a service?

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

# Angular Singleton Design pattern

**A singleton is a class that allows only a single instance to be created and gives access to that created instance**. It contains static variables that can accommodate unique and private instances of itself. It is used in scenarios when a user wants to restrict instantiation of a class to only one object.

A singleton service is a service instance that is shared across components.

## Providing a singleton service

There are two ways to make a service a singleton in Angular:

* Declare root for the value of the @Injectable() providedIn property
* Include the service in the AppModule or in a module that is only imported by the AppModule

# What is dependency injection in Angular?

Dependency Injection (DI) is a core concept of Angular 2+ and allows a class receive dependencies from another class.

# What is the purpose of async pipe?

**The async pipe subscribes to an Observable or Promise and returns the latest value it has emitted**. When a new value is emitted, the async pipe marks the component to be checked for changes. When the component gets destroyed, the async pipe unsubscribes automatically to avoid potential memory leaks.

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

You can store your component's template in one of two places. You can define it inline using the **template** property, or you can define the template in a separate HTML file and link to it in the component metadata using the **@Component** decorator's **templateUrl** property. The choice between inline and separate HTML is a matter of taste, circumstances, and organization policy. But normally we use inline template for small portion of code and external template file for bigger views. By default, the Angular CLI generates components with a template file. But you can override that with the below command,

ng generate component hero -it

# What is the purpose of ngFor directive?

We use Angular ngFor directive in the template to display each item in the list

# What happens if you use script tag inside template?

**Angular recognizes the value as unsafe and automatically sanitizes it, which removes the <script> tag but keeps safe content such as the text content of the <script> tag. This way it eliminates the risk of script injection attacks. If you still use it then it will be ignored and a warning appears in the browser console**. Let's take an example of innerHtml property binding which causes XSS vulnerability,

export class InnerHtmlBindingComponent {

// For example, a user/attacker-controlled value from a URL.

htmlSnippet = 'Template <script>alert("0wned")</script> <b>Syntax</b>';

}

# What is interpolation?

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

# What are template expressions?

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

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

# What are pipes?

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

# What is a parameterized pipe?

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

# How do you chain pipes?

You can chain pipes together in potentially useful combinations as per the needs. Let's take a birthday property which uses date pipe(along with parameter) and uppercase pipes as below

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

@Component({

selector: 'app-birthday',

template: `<p>Birthday is {{ birthday | date:'fullDate' | uppercase}} </p>` // THURSDAY, JUNE 18, 1987

})

export class BirthdayComponent {

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

}

# What is a custom pipe?

Apart from built-inn pipes, you can write your own custom pipe with the below key characteristics,

A pipe is a class decorated with pipe metadata **@Pipe** decorator, which you import from the core Angular library For example,

@Pipe({name: 'myCustomPipe'})

The pipe class implements the **PipeTransform** interface's transform method that accepts an input value followed by optional parameters and returns the transformed value. The structure of pipeTransform would be as below,

interface PipeTransform {

transform(value: any, ...args: any[]): any

}

The @Pipe decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier.

template: `{{someInputValue | myCustomPipe: someOtherValue}}`

# What is the difference between pure and impure pipe?

**A pure pipe is only called when Angular detects a change in the value or the parameters passed to a pipe**. For example, any changes to a primitive input value (String, Number, Boolean, Symbol) or a changed object reference (Date, Array, Function, Object). **An impure pipe is called for every change detection cycle no matter whether the value or parameters changes**. i.e., an impure pipe is called often, as often as every keystroke or mouse-move.

# What is a bootstrapping module?

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

/\* JavaScript imports \*/

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

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

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

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

# What are observables?

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

# What is HttpClient and its benefits?

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

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

The major advantages of HttpClient can be listed as below,

* Contains testability features
* Provides typed request and response objects
* Intercept request and response
* Supports Observalbe APIs
* Supports streamlined error handling

# How do you perform Error handling?

If the request fails on the server or failed to reach the server due to network issues then HttpClient will return an error object instead of a successful reponse. In this case, you need to handle in the component by **passing error object as a second callback to subscribe()** method. Let's see how it can be handled in the component with an example,

fetchUser() {

this.userService.getProfile()

.subscribe(

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

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

);

}

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

# What is RxJS?

RxJS is a library for composing asynchronous and callback-based code in a functional, reactive style using Observables. Many APIs such as HttpClient produce and consume RxJS Observables and also uses operators for processing observables. For example, you can import observables and operators for using HttpClient as below,

import { Observable, throwError } from 'rxjs';

import { catchError, retry } from 'rxjs/operators';

# What is subscribing?

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

# What is an observer?

Observer is an interface for a consumer of push-based notifications delivered by an Observable.

# What is the difference between promise and observable?

Below are the list of differences between promise and observable,

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

# How do you perform error handling in observables?

You can handle errors by specifying an **error callback** on the observer instead of relying on try/catch which are ineffective in asynchronous environment. For example, you can define error callback as below,

myObservable.subscribe({

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

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

});

# What are dynamic components?

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

# What is Angular Router?

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

# What is the purpose of base href tag?

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

<base href="/">

# What are the router imports?

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

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

# What is router outlet?

The RouterOutlet is a directive from the router library and it acts as a placeholder that marks the spot in the template where the router should display the components for that outlet. Router outlet is used like a component,

<router-outlet></router-outlet>

<!-- Routed components go here -->

# What are router links?

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

<h1>Angular Router</h1>

<nav>

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

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

</nav>

<router-outlet></router-outlet>

# What are active router links?

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

<h1>Angular Router</h1>

<nav>

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

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

</nav>

<router-outlet></router-outlet>

# What is router state?

RouterState is a tree of activated routes. Every node in this tree knows about the "consumed" URL segments, the extracted parameters, and the resolved data. You can access the current RouterState from anywhere in the application using the Router service and the routerState property.

@Component({templateUrl:'template.html'})

class MyComponent {

constructor(router: Router) {

const state: RouterState = router.routerState;

const root: ActivatedRoute = state.root;

const child = root.firstChild;

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

//...

}

}

# What are router events?

During each navigation, the Router emits navigation events through the Router.events property allowing you to track the lifecycle of the route. The sequence of router events is as below,

NavigationStart,

RouteConfigLoadStart,

RouteConfigLoadEnd,

RoutesRecognized,

GuardsCheckStart,

ChildActivationStart,

ActivationStart,

GuardsCheckEnd,

ResolveStart,

ResolveEnd,

ActivationEnd

ChildActivationEnd

NavigationEnd,

NavigationCancel,

NavigationError

Scroll

# What is activated route?

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

@Component({...})

class MyComponent {

constructor(route: ActivatedRoute) {

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

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

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

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

}

}

# What is the purpose of Wildcard route?

If the URL doesn't match any predefined routes then it causes the router to throw an error and crash the app. In this case, you can use wildcard route. A wildcard route has a path consisting of two asterisks to match every URL. For example, you can define PageNotFoundComponent for wildcard route as below

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

# Do I need a Routing Module always?

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

# What are different types of compilation in Angular?

Angular offers two ways to compile your application,

Just-in-Time (JIT)

Ahead-of-Time (AOT)

# What is JIT?

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

ng build

ng serve

# What is AOT?

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

ng build --aot

ng serve --aot

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

# What are the advantages with AOT?

Below are the list of AOT benefits,

**Faster rendering:** The browser downloads a pre-compiled version of the application. So it can render the application immediately without compiling the app.

**Fewer asynchronous requests:** It inlines external HTML templates and CSS style sheets within the application javascript which eliminates separate ajax requests.

**Smaller Angular framework download size:** Doesn't require downloading the Angular compiler. Hence it dramatically reduces the application payload.

**Detect template errors earlier:** Detects and reports template binding errors during the build step itself

**Better security:** It compiles HTML templates and components into JavaScript. So there won't be any injection attacks.

# What are the ways to control AOT compilation?

You can control your app compilation in two ways

By providing template compiler options in the tsconfig.json file

By configuring Angular metadata with decorators

# What is a service worker and its role in Angular?

A service worker is a script that runs in the web browser and manages caching for an application. Starting from 5.0.0 version, Angular ships with a service worker implementation. Angular service worker is designed to optimize the end user experience of using an application over a slow or unreliable network connection, while also minimizing the risks of serving outdated content.

# What are the design goals of service workers?

Below are the list of design goals of Angular's service workers,

It caches an application just like installing a native application

A running application continues to run with the same version of all files without any incompatible files

When you refresh the application, it loads the latest fully cached version

When changes are published then it immediately updates in the background

Service workers saves the bandwidth by downloading the resources only when they changed.

# What is Angular Ivy?

Angular Ivy is a new rendering engine for Angular. You can choose to opt in a preview version of Ivy from Angular version 8.

You can enable ivy in a new project by using the --enable-ivy flag with the ng new command

ng new ivy-demo-app --enable-ivy

You can add it to an existing project by adding enableIvy option in the angularCompilerOptions in your project's tsconfig.app.json.

{

"compilerOptions": { ... },

"angularCompilerOptions": {

"enableIvy": true

}

}

# What are the features included in ivy preview?

You can expect below features with Ivy preview,

Generated code that is easier to read and debug at runtime

Faster re-build time

Improved payload size

Improved template type checking

# Can I use AOT compilation with Ivy?

Yes, it is a recommended configuration. Also, AOT compilation with Ivy is faster. So you need set the default build options(with in angular.json) for your project to always use AOT compilation.

{

"projects": {

"my-project": {

"architect": {

"build": {

"options": {

...

"aot": true,

}

}

}

}

}

}

# How do you add web workers in your application?

You can add web worker anywhere in your application. For example, If the file that contains your expensive computation is src/app/app.component.ts, you can add a Web Worker using ng generate web-worker app command which will create src/app/app.worker.ts web worker file. This command will perform below actions,

Configure your project to use Web Workers

Adds app.worker.ts to receive messages

addEventListener('message', ({ data }) => {

const response = `worker response to ${data}`;

postMessage(response);

});

The component app.component.ts file updated with web worker file

if (typeof Worker !== 'undefined') {

// Create a new

const worker = new Worker('./app.worker', { type: 'module' });

worker.onmessage = ({ data }) => {

console.log('page got message: $\{data\}');

};

worker.postMessage('hello');

} else {

// Web Workers are not supported in this environment.

}

**Note:** You may need to refactor your initial scaffolding web worker code for sending messages to and from.

# What are the limitations with web workers?

You need to remember two important things when using Web Workers in Angular projects,

Some environments or platforms(like @angular/platform-server) used in Server-side Rendering, don't support Web Workers. In this case you need to provide a fallback mechanism to perform the computations to work in this environments.

Running Angular in web worker using @angular/platform-webworker is not yet supported in Angular CLI.

# What is Angular CLI Builder?

In Angular8, the CLI Builder API is stable and available to developers who want to customize the Angular CLI by adding or modifying commands. For example, you could supply a builder to perform an entirely new task, or to change which third-party tool is used by an existing command.

# What is declarable in Angular?

Declarable is a class type that you can add to the declarations list of an NgModule. The class types such as components, directives, and pipes comes can be declared in the module.

# what is an rxjs subject in Angular

An RxJS Subject is a special type of Observable that allows values to be multicasted to many Observers. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), Subjects are multicast.

A Subject is like an Observable, but can multicast to many Observers. Subjects are like EventEmitters: they maintain a registry of many listeners.

import { Subject } from 'rxjs';

const subject = new Subject<number>();

subject.subscribe({

next: (v) => console.log(`observerA: ${v}`)

});

subject.subscribe({

next: (v) => console.log(`observerB: ${v}`)

});

subject.next(1);

subject.next(2);

# What happens if I import the same module twice?

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

# How do you select an element with in a component template?

You can use @ViewChild directive to access elements in the view directly. Let's take input element with a reference,

<input #uname>

and define view child directive and access it in ngAfterViewInit lifecycle hook

@ViewChild('uname') input;

ngAfterViewInit() {

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

}

# How do you detect route change in Angular?

In Angular7, you can subscribe to router to detect the changes. The subscription for router events would be as below,

this.router.events.subscribe((event: Event) => {})

Let's take a simple component to detect router changes

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

import { Router, Event, NavigationStart, NavigationEnd, NavigationError } from '@angular/router';

@Component({

selector: 'app-root',

template: `<router-outlet></router-outlet>`

})

export class AppComponent {

constructor(private router: Router) {

this.router.events.subscribe((event: Event) => {

if (event instanceof NavigationStart) {

// Show loading indicator and perform an action

}

if (event instanceof NavigationEnd) {

// Hide loading indicator and perform an action

}

if (event instanceof NavigationError) {

// Hide loading indicator and perform an action

console.log(event.error); // It logs an error for debugging

}

});

}

}

# How do you pass headers for HTTP client?

You can directly pass object map for http client or create HttpHeaders class to supply the headers.

constructor(private \_http: HttpClient) {}

this.\_http.get('someUrl',{

headers: {'header1':'value1','header2':'value2'}

});

(or)

let headers = new HttpHeaders().set('header1', headerValue1); // create header object

headers = headers.append('header2', headerValue2); // add a new header, creating a new object

headers = headers.append('header3', headerValue3); // add another header

let params = new HttpParams().set('param1', value1); // create params object

params = params.append('param2', value2); // add a new param, creating a new object

params = params.append('param3', value3); // add another param

return this.\_http.get<any[]>('someUrl', { headers: headers, params: params })

# Is Angular supports dynamic imports?

Yes, Angular 8 supports dynamic imports in router configuration. i.e, You can use the import statement for lazy loading the module using loadChildren method and it will be understood by the IDEs(VSCode and WebStorm), webpack, etc. Previously, you have been written as below to lazily load the feature module. By mistake, if you have typo in the module name it still accepts the string and throws an error during build time.

{path: ‘user’, loadChildren: ‘./users/user.module#UserModulee’},

This problem is resolved by using dynamic imports and IDEs are able to find it during compile time itself.

{path: ‘user’, loadChildren: () => import(‘./users/user.module’).then(m => m.UserModule)};

# What is lazy loading?

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

const routes: Routes = [

{

path: 'customers',

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

},

{

path: 'orders',

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

},

{

path: '',

redirectTo: '',

pathMatch: 'full'

}

];

# How do you upgrade angular version?

The Angular upgrade is quite easier using Angular CLI ng update command as mentioned below. For example, if you upgrade from Angular 7 to 8 then your lazy loaded route imports will be migrated to the new import syntax automatically.

$ ng update @angular/cli @angular/core

# How to use polyfills in Angular application?

The Angular CLI provides support for polyfills officially. When you create a new project with the ng new command, a src/polyfills.ts configuration file is created as part of your project folder. This file includes the mandatory and many of the optional polyfills as JavaScript import statements. Let's categorize the polyfills,

**Mandatory polyfills:** These are installed automatically when you create your project with ng new command and the respective import statements enabled in 'src/polyfills.ts' file.

**Optional polyfills:** You need to install its npm package and then create import statement in 'src/polyfills.ts' file. For example, first you need to install below npm package for adding web animations (optional) polyfill.

npm install --save web-animations-js

and create import statement in polyfill file.

import 'web-animations-js';

# What are the ways to trigger change detection in Angular?

You can inject either ApplicationRef or NgZone, or ChangeDetectorRef into your component and apply below specific methods to trigger change detection in Angular. i.e, There are 3 possible ways,

* ApplicationRef.tick(): Invoke this method to explicitly process change detection and its side-effects. It check the full component tree.
* NgZone.run(callback): It evaluate the callback function inside the Angular zone.
* ChangeDetectorRef.detectChanges(): It detects only the components and it's children.

# What are the security principles in angular?

* You should avoid direct use of the DOM APIs.
* You should enable Content Security Policy (CSP) and configure your web server to return appropriate CSP HTTP headers.
* You should Use the offline template compiler.
* You should Use Server Side XSS protection.
* You should Use DOM Sanitizer.
* You should Preventing CSRF or XSRF attacks.