Angular

1) Overview:

Angular is a JavaScript framework which makes you able to create reactive **Single Page Applications** (SPAs). This is a leading front-end development framework which is regularly updated by Angular team of Google. Angular 7 is completely based on components. It consists of several components forming a tree structure with parent and child components.

2) Angular Version (updates)

Angular JS	Angular 2
The first version of Angular was released in the year of 2010. Some people call this as AngularJS and some people call as Angular 1. But it is officially named as AngularJS.	• Released in 2016 • Complete rewrite of Angular 1 • Written entirely in typescript • Component-based instead of Controller • ES6 and typescript supported • More testable as component-based • Support for Mobile/Low-end devices • Up to typescript 1.8 is supported
Angular 3	**Angular 4**
Why we don't have Angular 3? — Angular is being developed in a MonoRepo it means a single repo for everything. @angular/core, @angular/compiler, @angular/router etc are in the same repo and may have their own versions. — The angular router was already in v3 and releasing angular 3 with router 4 will create confusion — To avoid this confusion they decided to skip the version 3 and release with version 4.0.0 so that every major dependency in the MonoRepo are on the right track.	• Released in 2017 • Changes in core library • Angular 4 is simply the next version of angular 2, the underlying concept is the same & is an inheritance from Angular 2 • Lot of performance improvement is made to reduce size of AOT compiler generated code • Typescript 2.1 & 2.2 compatible — all feature of ts 2.1 & 2.2 are supported in Angular 4 application • Animation features are separated from @angular/core to @angular/animation — don't import @animation packages into the application to reduce bundle size and it gives the performance improvement. • Else block in *ngIf introduced: — Instead of writing 2 ngIf for else , simply add below code in component template: <ng-template *ngIf="yourCondition; else myFalsyTemplate"> <!--If HTML--> </ng-template> <ng-template #myFalsyTemplate>Else Html</ng-template>
Angular 5	**Angular 6**

• Released 1st November 2017
• Build optimizer: It helps to removed unnecessary code from your application
• Angular Universal State Transfer API and DOM Support — By using this feature, we can now share the state of the application between the server side and client side very easily.
• Compiler Improvements: This is one of the very nice features of Angular 5, which improved the support of incremental compilation of an application.
• Preserve White space: To remove unnecessary new lines, tabs and white spaces we can add below code(decrease bundle size)
// in component decorator you can now add:
"preserveWhitespaces: false"
// or in tsconfig.json:
"angularCompilerOptions": { "preserveWhitespaces": false}`
• Increased the standardization across all browsers: For internationalization we were depending on `i18n` , but in ng 5 provides a new date, number, and currency pipes which increases the internationalization across all the browsers and eliminates the need of i18n polyfills.
• exportAs: In Angular 5, multiple names support for both directives and components
• HttpClient: until Angualar 4.3 @angular/HTTP was been used which is now depreciated and in Angular 5 a new module called HttpClientModule is introduced which comes under @angular/common/http package.
• Few new Router Life-cycle Events being added in Angular 5: In Angular 5 few new life cycle events being added to the router and those are:
 	
ActivationStart, ActivationEnd, ChildActivationStart, ChildActivationEnd, GuardsCheckStart, GuardsCheckEnd, ResolveStart and ResolveEnd.
	• Released on April 2018
	• This release is focused less on the underlying framework, and more on tool-chain and on making it easier to move quickly with angular in the future
	• No major breaking changes
	• Dependency on RxJS 6 (this upgrade have breaking changes but CLI command helps in migrating from older version of RxJS)
	• Synchronizes major version number of the:
	    — Angular framework
	    — Angular CLI
	    — Angular Material + CDK
	• All of the above are now version 6.0.0, minor and patch releases though are completely independent and can be changed based on a specific project.
	• Remove support for <template> tag and "<ng-template>" should be used.
	• Registering provider: To register new service/provider, we import Service into module and then inject in provider array. e.g:
	
	// app.module.ts
	import {MyService} from './my-service';
	...
	providers: [...MyService]
	...
	But after this upgrade you will be able to add providedIn property in injectable decorator. e.g:
	// MyService.ts
	@Injectable({ providedIn: 'root'})
	export class MyService{}
	
	• ngModelChange event works well in this version
	
	• CLI Changes: Two new commands have been introduced
	— ng update <package>
	* Analyse package.json and recommend updates to your application
	* 3rd parties can provide update scripts using schematics
	* automatically update code for breaking changes
	* staying update and low maintenance
	— ng add

- Angular 5 supports TypeScript 2.3 version.
- Improved in faster Compiler support:
- A huge improvement made in an Angular compiler to make the development build faster. We can now take advantage of by running the below command in our development terminal window to make the build faster.
- ng serve/s — aot

  * add new capablities to your applicaiton
  * e.g ng add @angular/material : behind the scene it add bit of necessary code and changes project where needed to add it the thing we just told it to add.
  * Now adding things like angular material, progressive web app, service workers & angular elements to your existing ng application will be easy.
- CLI + Material starter templates: Let angular create code snippet for your basic components. e.g:
  — Material Sidenav
  * ng generate @angular/material:material-nav — name=my-nav
  Generate a starter template including a toolbar with app name and then the side navigation & it's also responsive
  — Dashboard
  * ng generate @angular/material:material-dashboard — name=my-dashboard
  Generates Dynamic list of cards
  — Datatable
  * ng generate @angular/material:material-table — name=my-table
  Generates Data Table with sorting, filtering & pagination
- It uses angular.json instead of .angular-cli.json
- Support for multiple projects: Now in angular.json we can add multiple projects
- initial release of Angular Elements which gives us ability to use our angular components in other environments like a Vue.js application. Its potential is truly amazing but unfortunately this release only works for angular application, we need to wait for next release to wrap out angular component into custom element and use it with framework like Vue.js

Angular 7

- Released on October 2018
- This is a major release and expanding to the entire platform including-
  - Core framework,
  - Angular Material,
  - CLI
- CLI Prompts: The CLI will now prompt users as when running common commands likeng new or ng add @angular/material with the intend of getting aid for building a new project using SCSS.
- Added a new interface — UrlSegment[] to CanLoad interface
- Added a new interface — DoBootstrap interface
- Angular 7 added a new compiler — Compatibility Compiler (ngcc)
- Introduce a new Pipe called — KeyValuePipe
- Angular 7 now supporting to TypeScript 2.9.
- Added a new elements features — enable Shadow DOM v1 and slots
- Added a new router features — warn if navigation triggered outside Angular zone
- Added a new mapping for ngfactory and ngsummary files to their module names in AOT summary resolver.
- Added a new “original” placeholder value on extracted XMB
- Added a new ability to recover from malformed URLs
- Added a new compiler support dot (.) in import statements and also avoid a crash in ngc-wrapped
- Update compiler to flatten nested template fns

Angular 8

- Releasing March/April 2019
- Being smaller, faster and easier to use and it will be making Angular developers life easier.
- Added Support for TypeScript 3.2
- Added a Navigation Type Available during Navigation in the Router
- Added pathParamsOrQueryParamsChange mode for runGuardsAndResolvers in the Router
- Allow passing state to routerLink Directives in the Router
- Allow passing state to NavigationExtras in the Router
- Restore the whole object when navigating back to a page managed by Angular Router
- Added support for SASS
- Resolve generated Sass/Less files to .css inputs
- Added Predicate function mode for runGuardsAndResolvers:-

This option means guards and resolvers will ignore changes when a provided predicate function returns `false`. This supports use cases where an application needs to ignore some param updates but not others. For example, changing a sort param in the URL might need to be ignored, whereas changing the `project` param might require a re-run of guards and resolvers.

- Added functionality to mark a control and its descendant controls as touched: — add markAllAsTouched () to AbstractControl
- Added an ng-new command that builds the project with Bazel
- Use image based cache for windows BuildKite
- Export NumberValueAccessor & RangeValueAccessor directives
- Use shared DomElementSchemaRegistry instance for improve performance of platform-server(@angular/platform-server):-

Right now the ServerRendererFactory2 creates a new instance of the DomElementSchemaRegistry for each and every request, which is quite costly (for the Tour of Heroes SSR example this takes around **15%** of the overall execution time)

- Now the Performance Improvements on the core, more consistent about “typeof checks”: -

When testing whether `value` is an object, use the ideal sequence of strictly not equal to `null` followed by `typeof value === ‘object’` consistently. Specifically, there’s no point in using double equal with `null` since `undefined` is ruled out by the `typeof` check. Also avoid the unnecessary ToBoolean check on `value.ngOnDestroy` in `hasOnDestroy()`, since the `typeof value.ngOnDestroy === ‘function’` will only let closures pass and all closures are truish (with the notable exception of `document.all`, but that shouldn’t be relevant for the `ngOnDestroy` hook)

- In the Compiler-CLI, expose ngtsc as a TscPlugin
- Restore whole object when navigating back to a page managed by Angular Router:-

This feature adds a few capabilities. First, when a `popstate` event fires the value of `history.state` will be read and passed into `NavigationStart`. In the past, only the `navigationId` would be passed here.
Additionally, `NavigationExtras` has a new public API called `state` which is **any object that will be stored as a value** in `history.state` on navigation. For example, the object `{foo: ‘bar’}` will be written to `history.state` here: -
`router.navigateByUrl(‘/simple’, {state: {foo: ‘bar’}});`

Angular 9

- Smaller bundle sizes and augmented performance
  To compile a component in View Engine, Angular needs information about all its declarable dependencies, their declarable dependencies, and so on. This means that Angular libraries cannot be AOT-compiled using View Engine.
  To compile a component in Ivy, Angular only needs information about the component itself, except for the name and package name of its declarable dependencies. Most notably, Ivy doesn't need metadata of any declarable dependencies to compile a component.The principle of locality means that in general we will see faster build times.
- Faster testing
- Better debugging
- Improved CSS class and style binding → `<div [style.--my-var]="myProperty || 'any value'"></div>`
- Improved type checking
- Improved build errors
- Improved build times, enabling AOT on by default
- Improved Internationalization →
  To mark the greeting for translation, add the i18n attribute to the <h1> tag.
  ```
  <h1 i18n>Hello i18n!</h1>
              or
  <h1 i18n="this is i18n description">Hello i18n!</h1>
              or
  <img [src]="logo" i18n-title title="Angular logo" />
  ```
- The Ivy compiler: The default use of the Ivy compiler is the most important feature of Angular 9, Ivy is what actually designed to solve the major problems of Angular i.e the performance and large file size
  ⇨ "angularCompilerOptions": {   "enableIvy": true } [tsconfig.json]
- Selector-less bindings support for Angular Ivy
- Support for TypeScript Diagnostics Format
- Support for more scopes in providedIn --
- A New Type-Safe TestBed.inject() Method Instead of TestBed.get()
- Improvements to differential loading
- AOT compilation everywhere
- Bundle sizes
- Globalisation
  - Additional provider scopes → `@Injectable({    providedIn: 'platform'  })  class MyService {...}`
-
- Improved developer experience
- New debugging API in development mode
- Strict mode
- Improved component and directive class inheritance
- Latest TypeScript versions -- 3.7
- Improved server-side rendering with Angular Universal
- Improved styling experience →we can use ngStyle and ngClass in HTML tags
- Stabel Bazel release as opt-in option
- Angular Components
- Lazy-loaded components

  entryComponents declarations are deprecated as they are no longer needed. Any Ivy component can be lazy loaded and dynamically rendered.This means that we can now lazy load and render a component without routing or Angular modules. However, in practice we have to use component render modules or feature render modules to link a component's template to its declarable dependencies.

- Changes with Angular Forms

  In version 9, <ngForm></ngForm> is no longer a valid selector to use while referencing an Angular form. You can use the <ng-form></ng-form> instead. Also the warning for using the removed form tag has been removed too.
  Secondly, the FormsModule.withConfig has been removed and you can now use the FormsModule directly.

  - ModuleWithProviders with generic Support

  This is mostly for library owners.version 9 introduced ModuleWithProviders<T> generic support

```
--- version 8----
@NgModule({ ...}) export class MyModule {
 static forRoot(config: SomeConfig): ModuleWithProviders<**SomeModule**> {
   return {
         ngModule: SomeModule,
         providers: [{ provide: SomeConfig, useValue: config }]
   };
 }
}
```

Angular 10

- **TypeScript** 3.9 is now featured, with support for TypeScript 3.8 having been removed.
- TypeScript runtime library **TSlib2**.0, and TypeScript static analysis tool **TSLint 6** is used.
- A compiler interface has been added that wraps the actual **ngtsc compiler**. The language service-specific compiler manages multiple typecheck files using the project interface, creating Scriptinfos as necessary.
- The browser configuration for new projects has been updated to exclude older, less-used browsers. Support is deprecated for Internet Explorer 9, Internet Explorer 10, and Internet Explorer Mobile.
- Angular Package Format no longer includes ESM5 or FESM5 bundles, saving download and install time when running yarn or npm install for Angular packages and libraries.
- For the compiler, name spans have been added for property reads and method calls.
- **EntryPointFinder**, a program-based **entry-point** finder, has been added that can be seeded from the imports in a program specified by a tsjconfig.json file. This is expected to be faster than the **DirectoryWalkerEntryPointFinder** when the active program only imports a small proportion of the installed entry points.
- **Autocompletion** is being removed from HTML entities, such as &amp, because of questionable value and a performance issue.
- Explicit mapping is being exposed from closure to devmode files. This feature is aimed at development tools that have to translate production build inputs into their devmode equivalents.
- In a breaking change, generic has been made mandatory for **ModuleWithProviders**. A generic type parameter has been required for the ModuleWithProviders pattern to work with the Ivy compilation and rendering pipeline, but before this commit, View Engine allowed the generic type to be omitted. If a developer is using ModuleWithProviders without a generic type, a version 10 migration will update the code. But if a developer is using **View Engine** and depending on a library that omits the generic type, a build error will be issued. In this case, ngcc will not help and the migration will only cover application code. The library author should be contacted to fix their library. As a workaround, skipLibChecks could be set to false in tsconfig or updating an app to use Ivy only.
- Type-checking performance improvements have been made to the compiler-cli.
- To improve performance, the computation of basePaths has been made lazy, so work is only done if needed in **TargetedEntryPointFinder**. Previously, basePaths was computed whenever the finder was instantiated, which was a waste of effort in the case when the targeted entry-point had already been processed.
- Merging of multiple translation files is supported. Previously, only one translation file per locale was permitted. Now users can specify multiple files per locale, and the transactions from each file will be merged by messaging ID.
- **Async locking timeouts** can be configured. This adds support for the ngcc.config.js file for setting the retryAttempts and retryDelay options for the **AsyncLocker**. An integration test adds a new check for a timeout and uses the ngcc.config.js to reduce the timeout time to prevent the test from taking too long.
- In a breaking change, warnings about unknown elements now are logged as errors. While this will not break an app, it might trip up tools that expect nothing to be logged via console.error.
- In another breaking change, any resolver that returns EMPTY will cancel navigation. To allow navigation to continue, developers must update the resolvers to update some value, such as **default!Empty**.
- The addition of dependency information and **ng-content selectors** to metadata. This proposed compiler feature would provide additional metadata useful for tools such as the Angular Language Service, offering the ability to provide suggestions for directives/components defined in libraries.
- Performance improvements, achieved by reducing the size of the entrypoint manifest and a caching technique in the manifest. In addition, caching of dependencies is done in the entrypoint manifest and read from there rather than being computed every time. Previously, even if an entrypoint did not need processing, ngcc (Angular Ivy compatibility compiler) would parse the files of the entrypoint to compute dependencies, which would take a lot of time for large_node modules.
- To improve ngcc performance, immediate reporting of a stale lock file is now allowed. In addition, a cached copy of a parsed tsconfig file is stored that can be reused if the tsconfig path is the same.
- For the router, the **CanLoad** guard now can return Urltree. A CanLoad guard returning Urltree cancels current navigation and redirects. This matches current behavior available to **CanActivate** guards that also has been added. This does not affect preloading. A CanLoad guard blocks any preloading; any routes with a CanLoad guard will not be preloaded and the guards will not be executed as part of preloading.
- Propagation of the correct value span in an ExpressionBinding of a microsyntax expression to ParsedProperty, which in turn would propagate the span to the template ASTs (both VE and Ivy). This proposal also is for the compiler.
- In a fix to the core, logic would be added to undecorated-class migration to decorate derived classes of undecorated classes that use Angular features.
- In a breaking change, Urlmatcher's type will reflect that it could always return null.
- For the service-worker, a fix has been put in for a situation in which there was a chance that the service worker will never register when there is a long-running task or recurring timeout.

- A number of bug fixes have been made including the compiler avoiding undefined expressions in a holey array and the core avoiding a migration error when a non-existent symbol is imported. There is also a workaround in the core for the Terser inlining bug. Another bug fix properly identifies modules affected by overrides in TestBed.
- Angular NPM no longer contains certain jsdoc comments to support the Closure Compiler's advanced optimizations. This is a breaking change. Support for Closure Compiler in packages has been experimental and broken for some time. Anyone who uses Closure Compiler is likely better off consuming Angular packages built from sources directly rather than consuming versions published on NPM. As a temporary workaround, users can consider using their current build pipeline with **Closure flag --compilation_level=SIMPLE.** This flag will ensure that the build pipeline produces buildable, runnable artifacts, at a cost of increased payload size due to advanced optimizations being disabled.

3) Angular application setup and prerequisite [Node, NPM, Angular Cli]

NVM

nvm is a version manager for node.js, switching will be easy if multiple node version need to maintain in future.

install nvm →install nvm
command –v nvm,→check version
nvm ls-remote:->list latest node version available to install
nvm ls→show your installed version of node
nvm install node or nvm install 12.0→install node.js
nvm use node or nvm use node 12.0

Node.js

Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside a web browser.
$ node –version //v12.15.0

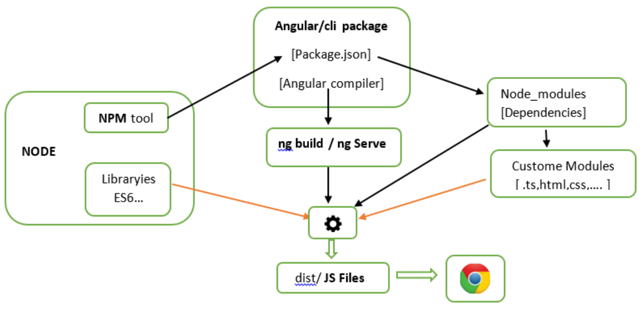
NPM

Npm is tool of node js which manages standard package for node which build node.js app easily.npm works with package.json file, it will install new libraries/dependancy for a project,it will register in package.json form command line or we can add manually.

$ npm install -g npm@latest →Install latest npm
$ npm list –g → list of global dependencies
$ npm list -g --depth=0 →List of package name only

$ npm init →create package.json file
npm outdated →Check for updates
$ npm install underscore →install dependency
$ npm install express momemt (install multiple depandancy)
$ npm uninstall underscore →create and install dependency
$ npm install underscore –save-dev
→create devDependancy propery
$ npm cache clean --force
$ npm audit fix
$ npm audit fix –force
$ npm audit fix
$ rm -R node_modules
$ npm install
→if you want to move application to other matching ,then first delete node module then
install .it will download all dependency
$ npm install -g @angular/cli
$ npm install -g @angular/cli@9.1.12
→It will install angular cli, with all default configuration
in package.json which is related to angular.

Angular Internal working with Node and NPM

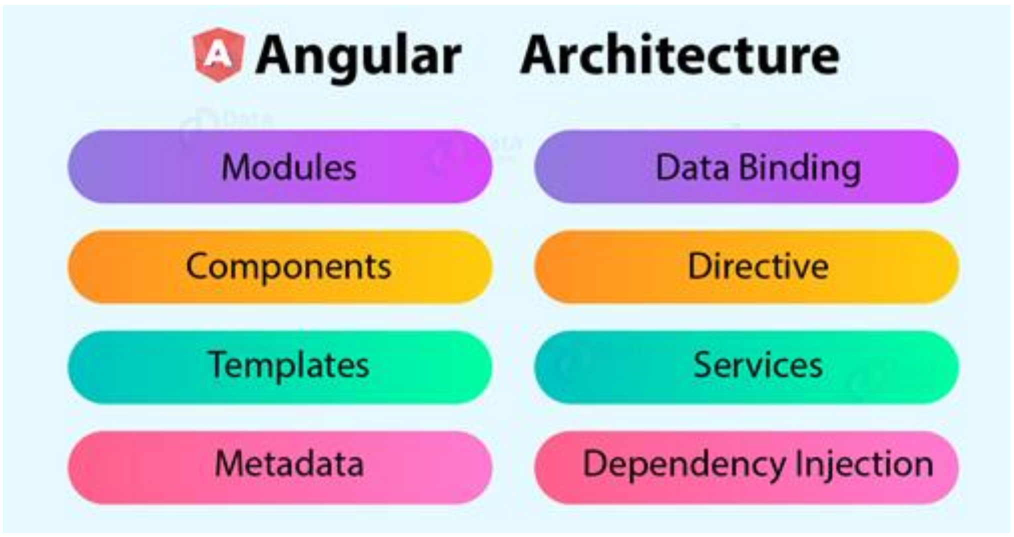


Angular cli commands

ng add	**It is used to add support for an external library to your project.**
ng build <project /lib> [option]	**It compiles an Angular app into an output directory named dist/ at the given output path. Must be executed from within a workspace directory.**
ng config	**It retrieves or sets Angular configuration values in the angular.json file for the workspace.**
ng Doc <keyword>	**It opens the official Angular documentation (angular.io) in a browser, and searches for a given keyword.**
ng e2e <project>	**It builds and serves an Angular app, then runs end-to-end tests using Protractor.**
ng generate <semantic> <name> [option]	**It generates and/or modifies files based on a schematic. Semantic lists:-** **appShell-generate appShell for server-side app,** **application-generate an application class,** **component,** **directive,** **enum,** **guard→generate guard** **interface,** **library→generate customer lib** **module→generate module** **pipe→generate custom pipe** **service** **serviceWorker,** **universal→** This command is used to pass this schematic to the "run" command to set up server-side rendering for an app
ng help	**It provides a list of available commands and their short descriptions.**
ng lint	**It is used to run linting tools on Angular app code in a given project folder.**
ng new	**It creates a new workspace and an initial Angular app.**
ng Run	**It runs an Architect target with an optional custom builder configuration defined in your project.**
ng serve	**It builds and serves your app, rebuilding on file changes.**
ng test	**It runs unit tests in a project.**
ng update	**It updates your application and its dependencies. See https://update.angular.io/**
ng version	**It utputs Angular CLI version.**
ng xi18n	**It extracts i18n messages from source code.**

4.) Angular Architecture

Various units that combine together to build an angular application. Angular tutorial is incomplete without its architecture and components:



Introduction to Angular concepts

Angular is a platform and framework for building single-page client applications using HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The architecture of an Angular application relies on certain fundamental concepts. The basic building blocks are *NgModules*, which provide a compilation context for *components*. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a *root module* that enables bootstrapping, and typically has many more *feature modules*.

- Components define *views*, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.
- Components use *services*, which provide specific functionality not directly related to views. Service providers can be *injected* into components as *dependencies*, making your code modular, reusable, and efficient.

Modules, components and services are classes that use *decorators*. These decorators mark their type and provide metadata that tells Angular how to use them.

- The metadata for a component class associates it with a *template* that defines a view. A template combines ordinary HTML with Angular *directives* and *binding markup* that allow Angular to modify the HTML before rendering it for display.
- The metadata for a service class provides the information Angular needs to make it available to components through *dependency injection (DI)*.

An app's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

Angular

- Angular provides advanced capabilities for mobile apps, animation, internationalization, server-side rendering, and more.
- Angular Material offers an extensive library of Material Design components.
- Angular Protractor offers an end-to-end testing framework for Angular apps.
- Angular also has an extensive network of 3rd-party tools and libraries.

1. Modules

Angular *NgModules* differ from and complement JavaScript (ES2015) modules. An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a *root module*, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

2. Components

Every Angular application has at least one component, the *root component* that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML *template* that defines a view to be displayed in a target environment.

The @Component() decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.

Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the Router NgModule.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of *lazy-loading*—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

3. Metadata

It is data about data. Decorators are metadata in angular.

4. Templates

A template combines HTML with Angular markup that can modify HTML elements before they are displayed.

Template directives *provide program logic, and* binding markup *connects your application data and the DOM.*

5. Data Binding

It is the synchronization between data and DOM. There are two kinds of Data Binding in Angular

- Event Binding
- Property Binding

6. Directive

Directives use for expanding the functionality of the HTML element.

Component vs directive

Directives use for expanding the functionality of the HTML element.
Components have their own view (HTML and styles). **Directives** are just "behavior" added to existing elements and **components**. **Component** extends **Directive** . Because of that there can only be one **component** on a host element, but multiple **directives**.

There are three kinds of directives in Angular:

1. Components—directives with a template.
2. Structural directives—
   change the DOM layout by adding and removing DOM elements,
   Built-in structural directives are ngIf,ngFor,ngSwitch
3. Attribute directives—
   change the appearance or behavior of an element, component, or another directive.
   Built-in Attribute directives are

- NgClass—adds and removes a set of CSS classes.
- NgStyle—adds and removes a set of HTML styles.
- NgModel—adds two-way data binding to an HTML form element.

7. Services

For data or logic that isn't associated with a specific view, and that you want to share across components, you create a *service* class. A service class definition is immediately preceded by the @Injectable() decorator. The decorator provides the metadata that allows other providers to be injected as dependencies into your class.

8. Dependency Injection

Dependency injection (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

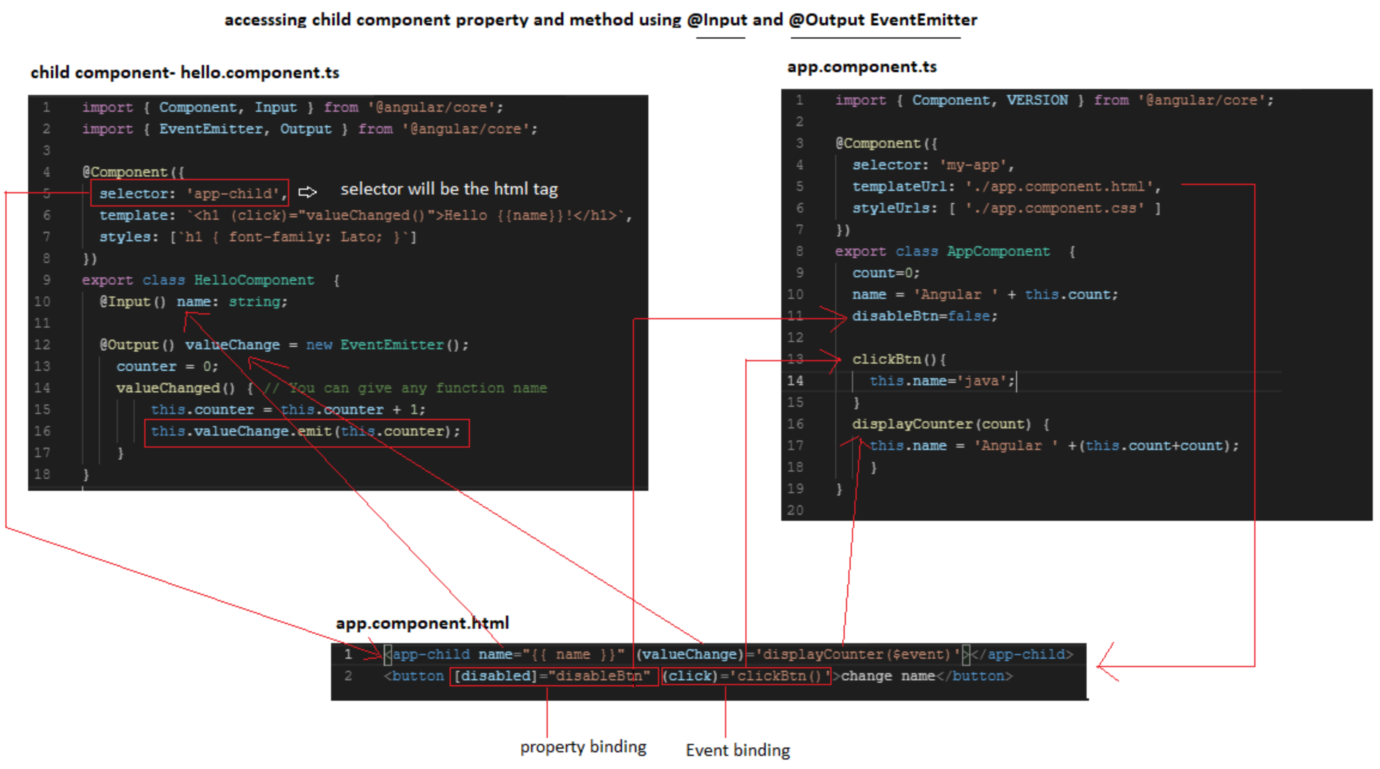
9.Routing

The Angular Router NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app. It is modeled on the familiar browser navigation conventions:

- Enter a URL in the address bar and the browser navigates to a corresponding page.
- Click links on the page and the browser navigates to a new page.
- Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

The router maps URL-like paths to views instead of pages. When a user performs an action, such as clicking a link, that would load a new page in the browser, the router intercepts the browser's behavior, and shows or hides view hierarchies.

If the router determines that the current application state requires particular functionality, and the module that defines it hasn't been loaded, the router can *lazy-load* the module on demand.


accesssing child component property and method using @Input and @Output EventEmitter

child component- hello.component.ts

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

@Component({
  selector: 'app-child',
  template: `<h1 (click)="valueChanged()">Hello {{name}}!</h1>`,
  styles: [`h1 { font-family: Lato; }`]
})
export class HelloComponent  {
  @Input() name: string;

  @Output() valueChange = new EventEmitter();
    counter = 0;
    valueChanged() { // You can give any function name
        this.counter = this.counter + 1;
        this.valueChange.emit(this.counter);
    }
}
```

app.component.ts

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

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent  {
  count=0;
  name = 'Angular ' + this.count;
  disableBtn=false;

  clickBtn(){
    this.name='java';
  }
  displayCounter(count) {
    this.name = 'Angular ' +(this.count+count);
    }
}
```

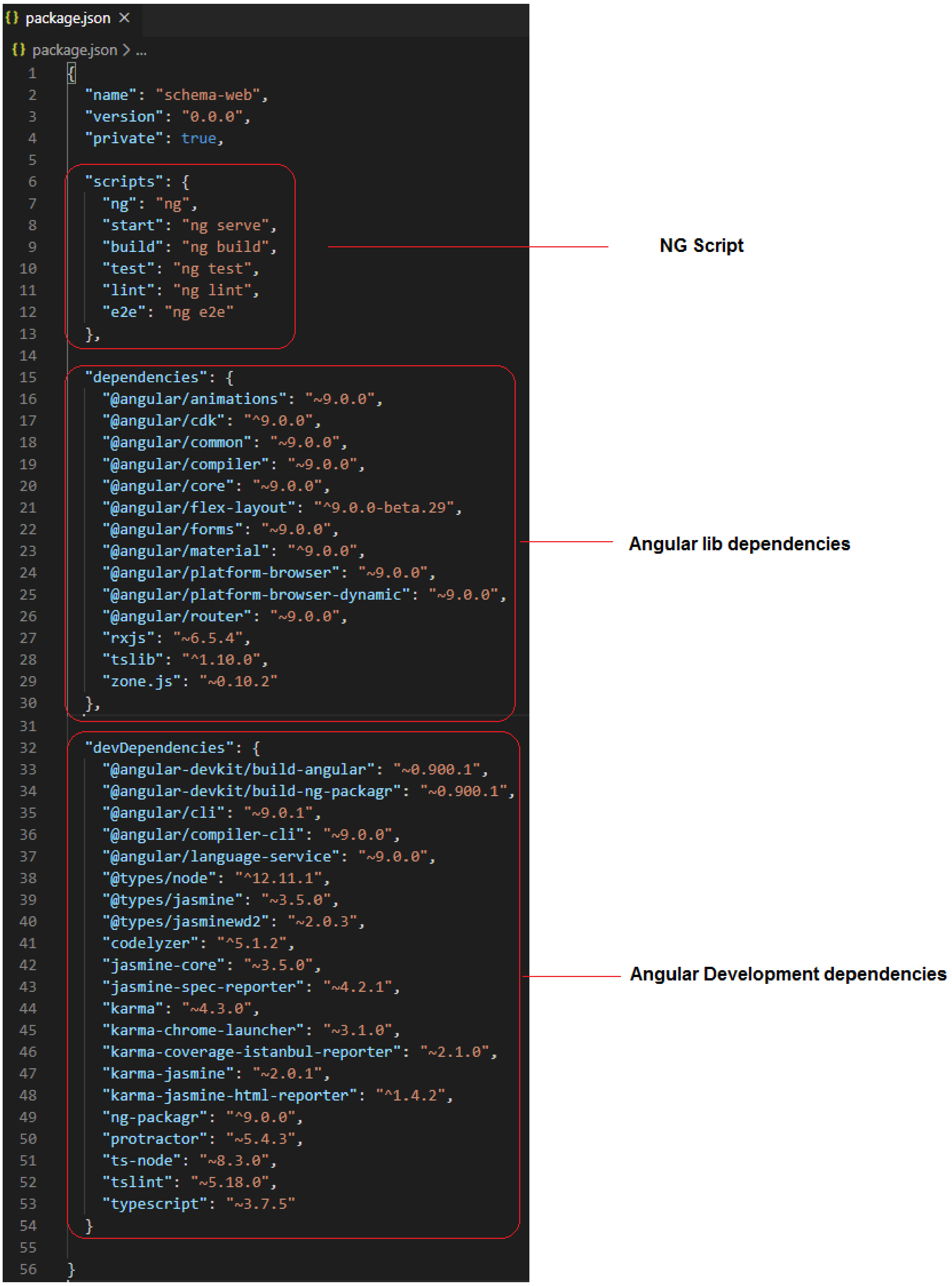
app.component.html

```
<app-child name="{{ name }}" (valueChange)='displayCounter($event)'></app-child>
<button [disabled]="disableBtn" (click)='clickBtn()'>change name</button>
```

5) Applications configuration files

1.Package.json

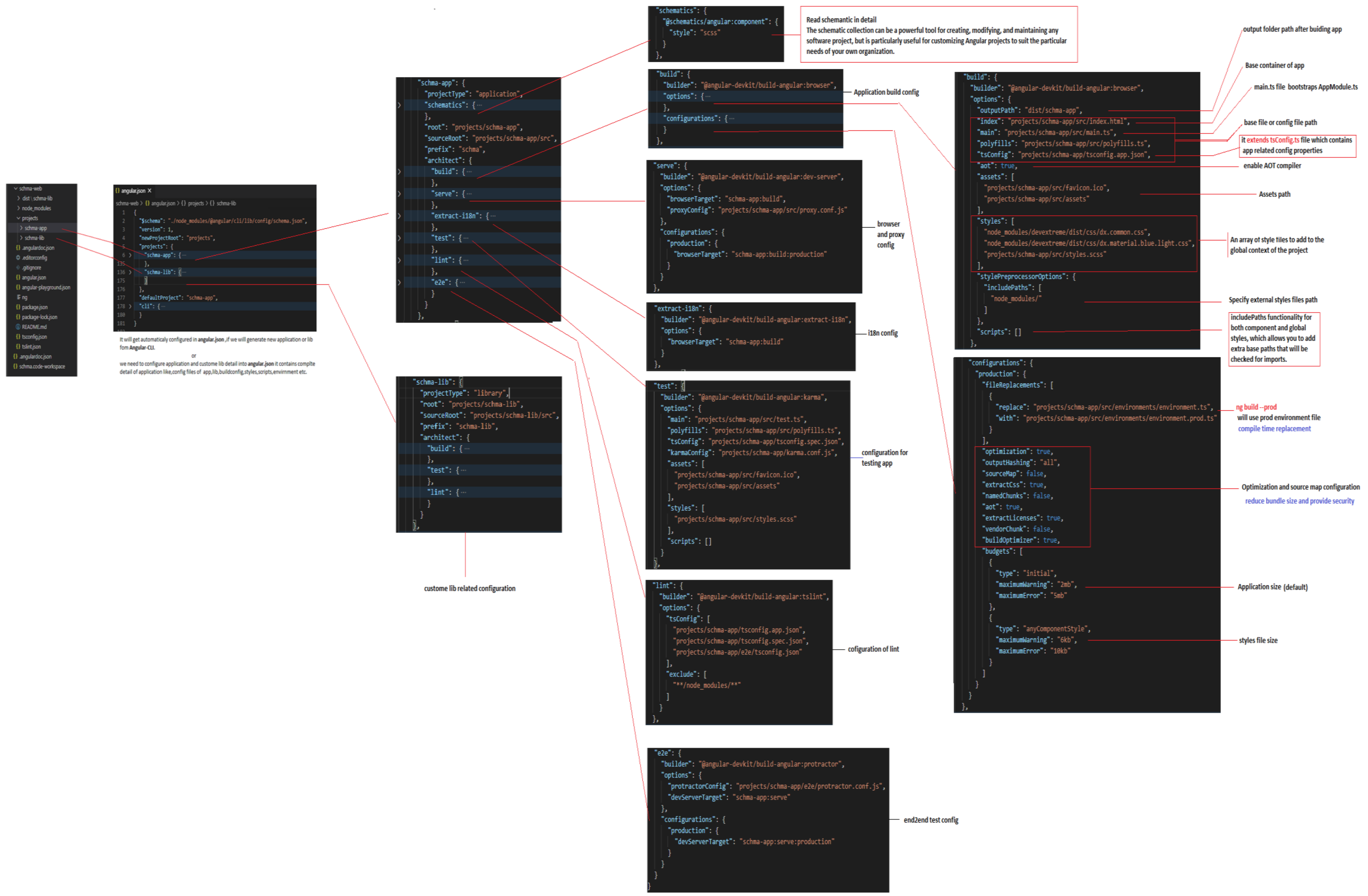
All npm packages contain a file, usually in the project root, called **package**. **json** - this file holds various metadata relevant to the project. This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies.



```json
{
  "name": "schema-web",
  "version": "0.0.0",
  "private": true,

  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },

  "dependencies": {
    "@angular/animations": "~9.0.0",
    "@angular/cdk": "^9.0.0",
    "@angular/common": "~9.0.0",
    "@angular/compiler": "~9.0.0",
    "@angular/core": "~9.0.0",
    "@angular/flex-layout": "^9.0.0-beta.29",
    "@angular/forms": "~9.0.0",
    "@angular/material": "^9.0.0",
    "@angular/platform-browser": "~9.0.0",
    "@angular/platform-browser-dynamic": "~9.0.0",
    "@angular/router": "~9.0.0",
    "rxjs": "~6.5.4",
    "tslib": "^1.10.0",
    "zone.js": "~0.10.2"
  },

  "devDependencies": {
    "@angular-devkit/build-angular": "~0.900.1",
    "@angular-devkit/build-ng-packagr": "~0.900.1",
    "@angular/cli": "~9.0.1",
    "@angular/compiler-cli": "~9.0.0",
    "@angular/language-service": "~9.0.0",
    "@types/node": "^12.11.1",
    "@types/jasmine": "~3.5.0",
    "@types/jasminewd2": "~2.0.3",
    "codelyzer": "^5.1.2",
    "jasmine-core": "~3.5.0",
    "jasmine-spec-reporter": "~4.2.1",
    "karma": "~4.3.0",
    "karma-chrome-launcher": "~3.1.0",
    "karma-coverage-istanbul-reporter": "~2.1.0",
    "karma-jasmine": "~2.0.1",
    "karma-jasmine-html-reporter": "^1.4.2",
    "ng-packagr": "^9.0.0",
    "protractor": "~5.4.3",
    "ts-node": "~8.3.0",
    "tslint": "~5.18.0",
    "typescript": "~3.7.5"
  }

}
```

2. angular.json (angular project setup detail)

A file named angular.json at the root level of an Angular workspace ,provides workspace- wide and project-specific configuration defaults for build and development tools provided by the Angular CLI. Path values given in the configuration are relative to the root workspace folder.



```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "schma-app": {…
    },
    "schma-lib": {…
    }
  },
  "defaultProject": "schma-app",
  "cli": {…
  }
}
```

It will get automaticaly configured in **angular.json** ,if we will generate new application or lib fom **Angular-CLI**.

or

we need to configure application and custome lib detail into **angular.json** it contains complte detail of application like,config files of app,lib,buildconfig,styles,scripts,envirnment etc.

```
"schma-app": {
  "projectType": "application",
  "schematics": {…
  },
  "root": "projects/schma-app",
  "sourceRoot": "projects/schma-app/src",
  "prefix": "schma",
  "architect": {
    "build": {…
    },
    "serve": {…
    },
    "extract-i18n": {…
    },
    "test": {…
    },
    "lint": {…
    },
    "e2e": {…
    }
  }
},
```

```
"schma-lib": {
  "projectType": "library",
  "root": "projects/schma-lib",
  "sourceRoot": "projects/schma-lib/src",
  "prefix": "schma-lib",
  "architect": {
    "build": {…
    },
    "test": {…
    },
    "lint": {…
    }
  }
}
```

custome lib related configuration

```
"schematics": {
  "@schematics/angular:component": {
    "style": "scss"
  }
},
```

Read schemantic in detail
The schematic collection can be a powerful tool for creating, modifying, and maintaining any software project, but is particularly useful for customizing Angular projects to suit the particular needs of your own organization.

```
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {…
  },
  "configurations": {…
  }
},
```

Application build config

```
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "options": {
    "browserTarget": "schma-app:build",
    "proxyConfig": "projects/schma-app/src/proxy.conf.js"
  },
  "configurations": {
    "production": {
      "browserTarget": "schma-app:build:production"
    }
  }
},
```

browser and proxy config

```
"extract-i18n": {
  "builder": "@angular-devkit/build-angular:extract-i18n",
  "options": {
    "browserTarget": "schma-app:build"
  }
},
```

i18n config

```
"test": {
  "builder": "@angular-devkit/build-angular:karma",
  "options": {
    "main": "projects/schma-app/src/test.ts",
    "polyfills": "projects/schma-app/src/polyfills.ts",
    "tsConfig": "projects/schma-app/tsconfig.spec.json",
    "karmaConfig": "projects/schma-app/karma.conf.js",
    "assets": [
      "projects/schma-app/src/favicon.ico",
      "projects/schma-app/src/assets"
    ],
    "styles": [
      "projects/schma-app/src/styles.scss"
    ],
    "scripts": []
  }
},
```

configuration for testing app

```
"lint": {
  "builder": "@angular-devkit/build-angular:tslint",
  "options": {
    "tsConfig": [
      "projects/schma-app/tsconfig.app.json",
      "projects/schma-app/tsconfig.spec.json",
      "projects/schma-app/e2e/tsconfig.json"
    ],
    "exclude": [
      "**/node_modules/**"
    ]
  }
},
```

cofiguration of lint

```
"e2e": {
  "builder": "@angular-devkit/build-angular:protractor",
  "options": {
    "protractorConfig": "projects/schma-app/e2e/protractor.conf.js",
    "devServerTarget": "schma-app:serve"
  },
  "configurations": {
    "production": {
      "devServerTarget": "schma-app:serve:production"
    }
  }
}
```

end2end test config

```
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
    "outputPath": "dist/schma-app",
    "index": "projects/schma-app/src/index.html",
    "main": "projects/schma-app/src/main.ts",
    "polyfills": "projects/schma-app/src/polyfills.ts",
    "tsConfig": "projects/schma-app/tsconfig.app.json",
    "aot": true,
    "assets": [
      "projects/schma-app/src/favicon.ico",
      "projects/schma-app/src/assets"
    ],
    "styles": [
      "node_modules/devextreme/dist/css/dx.common.css",
      "node_modules/devextreme/dist/css/dx.material.blue.light.css",
      "projects/schma-app/src/styles.scss"
    ],
    "stylePreprocessorOptions": {
      "includePaths": [
        "node_modules/"
      ]
    },
    "scripts": []
  },
```

output folder path after buiding app

Base container of app

main.ts file bootstraps AppModule.ts

base file or config file path

it extends tsConfig.ts file which contains app related config properties

enable AOT compiler

Assets path

An array of style files to add to the global context of the project

Specify external styles files path

includePaths functionality for both component and global styles, which allows you to add extra base paths that will be checked for imports.

```
  "configurations": {
    "production": {
      "fileReplacements": [
        {
          "replace": "projects/schma-app/src/environments/environment.ts",
          "with": "projects/schma-app/src/environments/environment.prod.ts"
        }
      ],
      "optimization": true,
      "outputHashing": "all",
      "sourceMap": false,
      "extractCss": true,
      "namedChunks": false,
      "aot": true,
      "extractLicenses": true,
      "vendorChunk": false,
      "buildOptimizer": true,
      "budgets": [
        {
          "type": "initial",
          "maximumWarning": "2mb",
          "maximumError": "5mb"
        },
        {
          "type": "anyComponentStyle",
          "maximumWarning": "6kb",
          "maximumError": "10kb"
        }
      ]
    }
  }
},
```

ng build --prod
will use prod environment file
compile time replacement

Optimization and source map configuration
reduce bundle size and provide security

Application size (default)

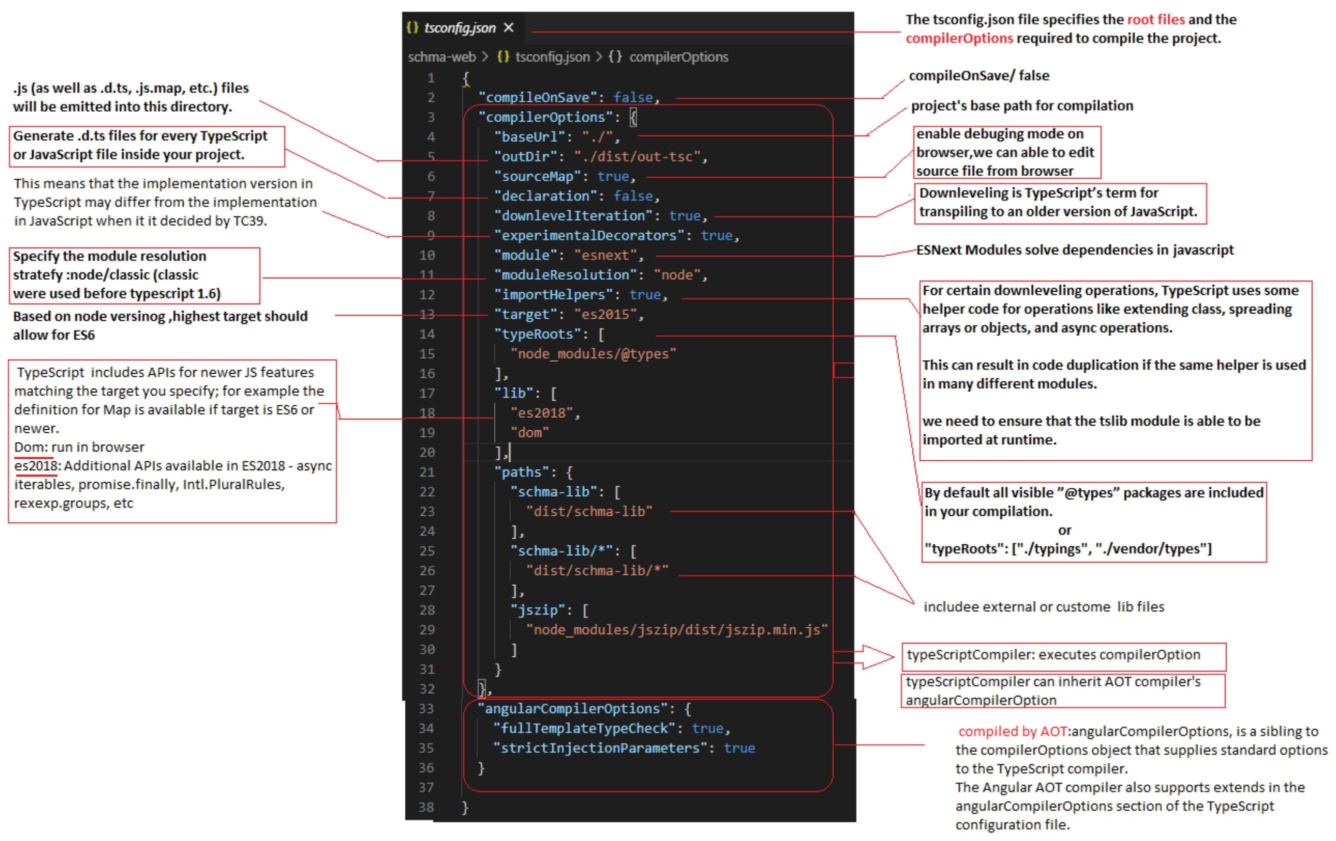
styles file size

3. tsconfig.json (setup detail)

When you use AOT compilation, you can control how your application is compiled by specifying *template* compiler options in the TypeScript configuration file (**tsconfig.json**).

The template options object, `angularCompilerOptions`, is a sibling to the `compilerOptions` object that supplies standard options to the TypeScript compiler.

note: there are several tsConfig properties ,this file contains basic properties.



```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "module": "esnext",
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "es2015",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2018",
      "dom"
    ],
    "paths": {
      "schma-lib": [
        "dist/schma-lib"
      ],
      "schma-lib/*": [
        "dist/schma-lib/*"
      ],
      "jszip": [
        "node_modules/jszip/dist/jszip.min.js"
      ]
    }
  },
  "angularCompilerOptions": {
    "fullTemplateTypeCheck": true,
    "strictInjectionParameters": true
  }
}
```

The tsconfig.json file specifies the root files and the compilerOptions required to compile the project.

compileOnSave/ false

project's base path for compilation

.js (as well as .d.ts, .js.map, etc.) files will be emitted into this directory.

enable debuging mode on browser,we can able to edit source file from browser

Generate .d.ts files for every TypeScript or JavaScript file inside your project.

Downleveling is TypeScript's term for transpiling to an older version of JavaScript.

This means that the implementation version in TypeScript may differ from the implementation in JavaScript when it it decided by TC39.

ESNext Modules solve dependencies in javascript

Specify the module resolution stratefy :node/classic (classic were used before typescript 1.6)

For certain downleveling operations, TypeScript uses some helper code for operations like extending class, spreading arrays or objects, and async operations.

This can result in code duplication if the same helper is used in many different modules.

we need to ensure that the tslib module is able to be imported at runtime.

Based on node versinog ,highest target should allow for ES6

TypeScript includes APIs for newer JS features matching the target you specify; for example the definition for Map is available if target is ES6 or newer.
Dom: run in browser
es2018: Additional APIs available in ES2018 - async iterables, promise.finally, Intl.PluralRules, rexexp.groups, etc

By default all visible "@types" packages are included in your compilation.
or
"typeRoots": ["./typings", "./vendor/types"]

includee external or custome lib files

typeScriptCompiler: executes compilerOption

typeScriptCompiler can inherit AOT compiler's angularCompilerOption

compiled by AOT:angularCompilerOptions, is a sibling to the compilerOptions object that supplies standard options to the TypeScript compiler.
The Angular AOT compiler also supports extends in the angularCompilerOptions section of the TypeScript configuration file.

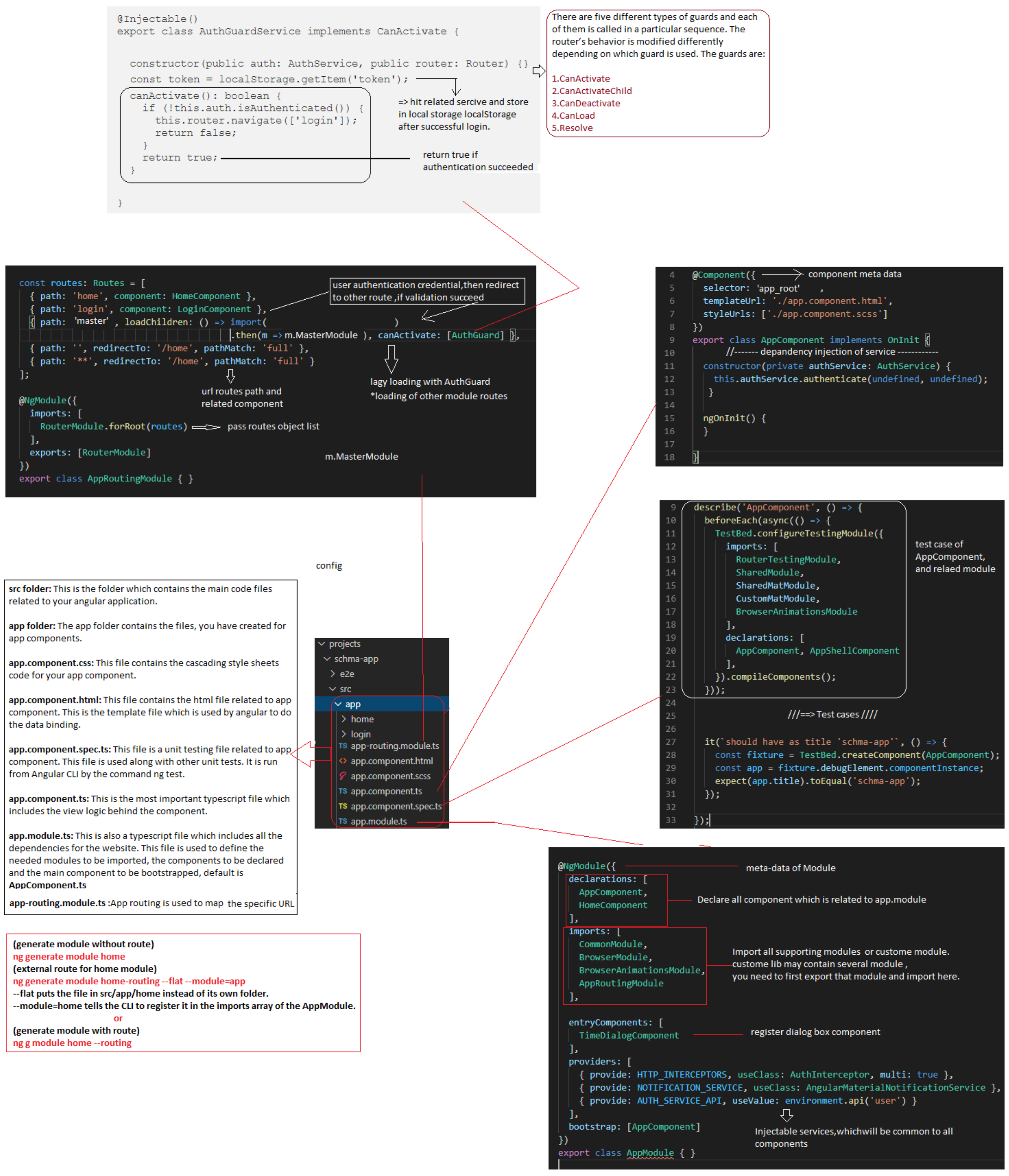
6) Application Basic structure

1.App.Module.ts (Basic setup)

Angular apps are modular and Angular has its own modularity system called *NgModules*. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular app has at least one NgModule class, the *root module*, which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by *bootstrapping* the root NgModule.

App Module setup



```
@Injectable()
export class AuthGuardService implements CanActivate {

  constructor(public auth: AuthService, public router: Router) {}
  const token = localStorage.getItem('token');
  canActivate(): boolean {
    if (!this.auth.isAuthenticated()) {
      this.router.navigate(['login']);
      return false;
    }
    return true;
  }

}
```

=> hit related sercive and store in local storage localStorage after successful login.

return true if authentication succeeded

There are five different types of guards and each of them is called in a particular sequence. The router's behavior is modified differently depending on which guard is used. The guards are:

1.CanActivate
2.CanActivateChild
3.CanDeactivate
4.CanLoad
5.Resolve

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'master' , loadChildren: () => import(                    )
                                        .then(m => m.MasterModule ), canActivate: [AuthGuard] },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', redirectTo: '/home', pathMatch: 'full' }
];

@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

user authentication credential,then redirect to other route ,if validation succeed

url routes path and related component

lagy loading with AuthGuard
*loading of other module routes

pass routes object list

m.MasterModule

```
@Component({
  selector: 'app_root'    ,
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
       //------- depandency injection of service -----------
  constructor(private authService: AuthService) {
    this.authService.authenticate(undefined, undefined);
   }

  ngOnInit() {
  }

}
```

component meta data

```
describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [
        RouterTestingModule,
        SharedModule,
        SharedMatModule,
        CustomMatModule,
        BrowserAnimationsModule
      ],
      declarations: [
        AppComponent, AppShellComponent
      ],
    }).compileComponents();
  }));

                ///==> Test cases ////

  it(`should have as title 'schma-app'`, () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app.title).toEqual('schma-app');
  });

});
```

test case of AppComponent, and relaed module

config

src folder: This is the folder which contains the main code files related to your angular application.

app folder: The app folder contains the files, you have created for app components.

app.component.css: This file contains the cascading style sheets code for your app component.

app.component.html: This file contains the html file related to app component. This is the template file which is used by angular to do the data binding.

app.component.spec.ts: This file is a unit testing file related to app component. This file is used along with other unit tests. It is run from Angular CLI by the command ng test.

app.component.ts: This is the most important typescript file which includes the view logic behind the component.

app.module.ts: This is also a typescript file which includes all the dependencies for the website. This file is used to define the needed modules to be imported, the components to be declared and the main component to be bootstrapped, default is **AppComponent.ts**

app-routing.module.ts :App routing is used to map the specific URL

(generate module without route)
ng generate module home
(external route for home module)
ng generate module home-routing --flat --module=app
--flat puts the file in src/app/home instead of its own folder.
--module=home tells the CLI to register it in the imports array of the AppModule.
or
(generate module with route)
ng g module home --routing

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent
  ],
  imports: [
    CommonModule,
    BrowserModule,
    BrowserAnimationsModule,
    AppRoutingModule
  ],

  entryComponents: [
    TimeDialogComponent
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
    { provide: NOTIFICATION_SERVICE, useClass: AngularMaterialNotificationService },
    { provide: AUTH_SERVICE_API, useValue: environment.api('user') }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

meta-data of Module

Declare all component which is related to app.module

Import all supporting modules or custome module.
custome lib may contain several module ,
you need to first export that module and import here.

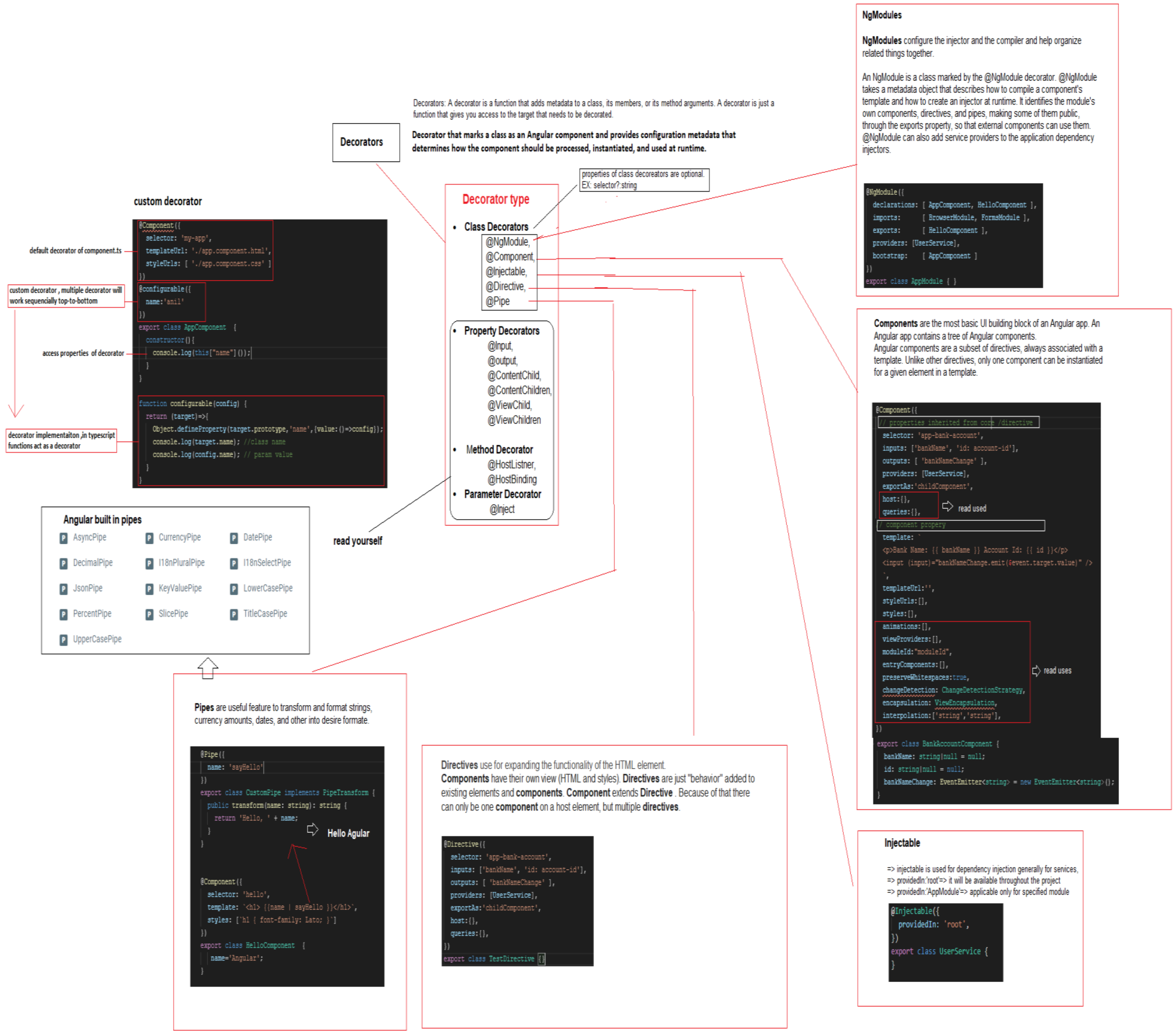
register dialog box component

Injectable services,whichwill be common to all components

2. Angular Decorators

Decorators are a design pattern that is used to separate modification or decoration of a class without modifying the original source code.
Decorators are functions that are invoked with a prefixed @ symbol, and immediately followed by a class , parameter, method or property.

Angular Decorators
- Class Decorators→ @NgModule,@Component,@Injectable,@Directive,@Pipe
- Property Decorators→@Input,@output, @ContentChild, @ContentChildren, @ViewChild, @ViewChildren
- Method Decorators→@HostListner, @HostBinding
- Parameter Decorators→@Inject



Decorators: A decorator is a function that adds metadata to a class, its members, or its method arguments. A decorator is just a function that gives you access to the target that needs to be decorated.

Decorator that marks a class as an Angular component and provides configuration metadata that determines how the component should be processed, instantiated, and used at runtime.

```
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
@configurable({
  name:'anil'
})
export class AppComponent  {
  constructor(){
    console.log(this["name"]());
  }
}

function configurable(config) {
  return (target)=>{
    Object.defineProperty(target.prototype,'name',{value:()=>config});
    console.log(target.name); //class name
    console.log(config.name); // param value
  }
}
```

Pipes are useful feature to transform and format strings, currency amounts, dates, and other into desire formate.

```
@Pipe({
  name: 'sayHello'
})
export class CustomPipe implements PipeTransform {
  public transform(name: string): string {
    return 'Hello, ' + name;
  }
}


@Component({
  selector: 'hello',
  template: `<h1> {{name | sayHello }}</h1>`,
  styles: [`h1 { font-family: Lato; }`]
})
export class HelloComponent  {
   name='Angular';
}
```

Directives use for expanding the functionality of the HTML element.
Components have their own view (HTML and styles). **Directives** are just "behavior" added to existing elements and **components**. **Component** extends **Directive** . Because of that there can only be one **component** on a host element, but multiple **directives**.

```
@Directive({
  selector: 'app-bank-account',
  inputs: ['bankName', 'id: account-id'],
  outputs: [ 'bankNameChange' ],
  providers: [UserService],
  exportAs:'childComponent',
  host:{},
  queries:{},
})
export class TestDirective {}
```

NgModules

NgModules configure the injector and the compiler and help organize related things together.

An NgModule is a class marked by the @NgModule decorator. @NgModule takes a metadata object that describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes, making some of them public, through the exports property, so that external components can use them. @NgModule can also add service providers to the application dependency injectors.

```
@NgModule({
  declarations: [ AppComponent, HelloComponent ],
  imports:      [ BrowserModule, FormsModule ],
  exports:      [ HelloComponent ],
  providers: [UserService],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Components are the most basic UI building block of an Angular app. An Angular app contains a tree of Angular components.
Angular components are a subset of directives, always associated with a template. Unlike other directives, only one component can be instantiated for a given element in a template.

```
@Component({
 // properties inherited from core /directive
  selector: 'app-bank-account',
  inputs: ['bankName', 'id: account-id'],
  outputs: [ 'bankNameChange' ],
  providers: [UserService],
  exportAs:'childComponent',
  host:{},
  queries:{},
 // component propery
  template: `
  <p>Bank Name: {{ bankName }} Account Id: {{ id }}</p>
  <input (input)="bankNameChange.emit($event.target.value)" />
  `,
  templateUrl:'',
  styleUrls:[],
  styles:[],
  animations:[],
  viewProviders:[],
  moduleId:"moduleId",
  entryComponents:[],
  preserveWhitespaces:true,
  changeDetection: ChangeDetectionStrategy,
  encapsulation: ViewEncapsulation,
  interpolation:['string','string'],
})
export class BankAccountComponent {
  bankName: string|null = null;
  id: string|null = null;
  bankNameChange: EventEmitter<string> = new EventEmitter<string>();
}
```

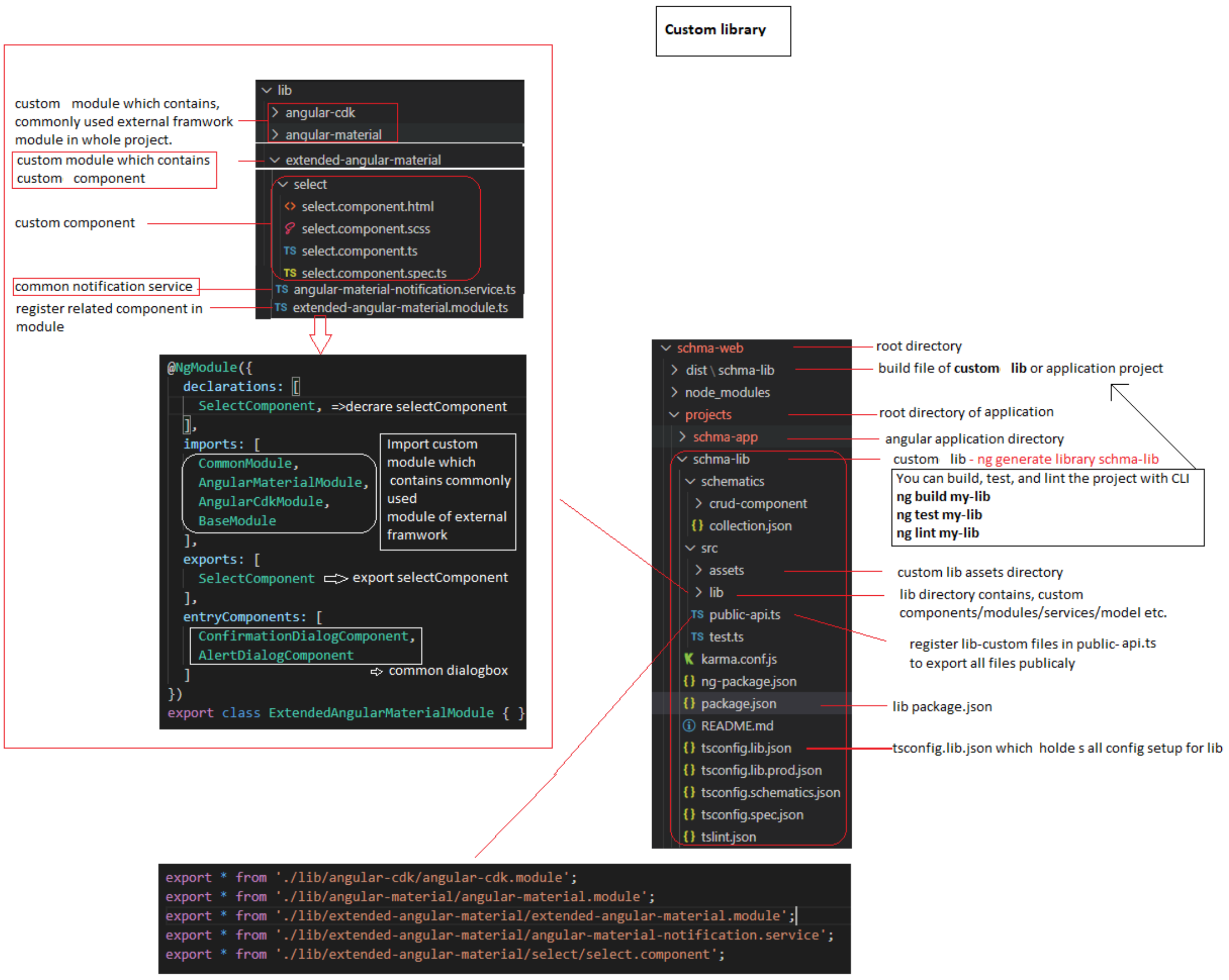
Injectable

=> injectable is used for dependency injection generally for services.
=> providedIn:'root'=> it will be available throughout the project
=> providedIn:'AppModule'=> applicable only for specified module

```
@Injectable({
  providedIn: 'root',
})
export class UserService {
}
```

3. Angular Custom Library

We can create and publish new libraries to extend Angular functionality. If you find that you need to solve the same problem in more than one app (or want to share your solution with other developers), you have a candidate for a library.


Custom library
custom module which contains, commonly used external framwork module in whole project.
custom module which contains custom component
custom component
common notification service
register related component in module
lib
angular-cdk
angular-material
extended-angular-material
select
select.component.html
select.component.scss
select.component.ts
select.component.spec.ts
angular-material-notification.service.ts
extended-angular-material.module.ts
=>decrare selectComponent
Import custom module which contains commonly used module of external framwork
export selectComponent
common dialogbox
schma-web
dist \ schma-lib
node_modules
projects
schma-app
schma-lib
schematics
crud-component
collection.json
src
assets
lib
public-api.ts
test.ts
karma.conf.js
ng-package.json
package.json
README.md
tsconfig.lib.json
tsconfig.lib.prod.json
tsconfig.schematics.json
tsconfig.spec.json
tslint.json
root directory
build file of custom lib or application project
root directory of application
angular application directory
custom lib - ng generate library schma-lib
You can build, test, and lint the project with CLI
ng build my-lib
ng test my-lib
ng lint my-lib
custom lib assets directory
lib directory contains, custom components/modules/services/model etc.
register lib-custom files in public- api.ts to export all files publicaly
lib package.json
tsconfig.lib.json which holde s all config setup for lib

```
@NgModule({
  declarations: [
    SelectComponent,
  ],
  imports: [
    CommonModule,
    AngularMaterialModule,
    AngularCdkModule,
    BaseModule
  ],
  exports: [
    SelectComponent
  ],
  entryComponents: [
    ConfirmationDialogComponent,
    AlertDialogComponent
  ]
})
export class ExtendedAngularMaterialModule { }
```

```
export * from './lib/angular-cdk/angular-cdk.module';
export * from './lib/angular-material/angular-material.module';
export * from './lib/extended-angular-material/extended-angular-material.module';
export * from './lib/extended-angular-material/angular-material-notification.service';
export * from './lib/extended-angular-material/select/select.component';
```

Dist folder:

Dist folder is the code that is actually deployed on a website. It is different from src code in the way that it is actually shorter in size because it doesn't have any comments and spaces,etc. We deploy dist folder instead of src folder because it takes less time for a browser to load a web page/web app that is smaller in size.

4. Angular FORMS

Handling user input with forms is the cornerstone of many common applications. Applications use forms to enable users to log in, to update a profile, to enter sensitive information, and to perform many other data-entry tasks.
There are two type of form

- **Reactive forms**
- **Template-driven forms**

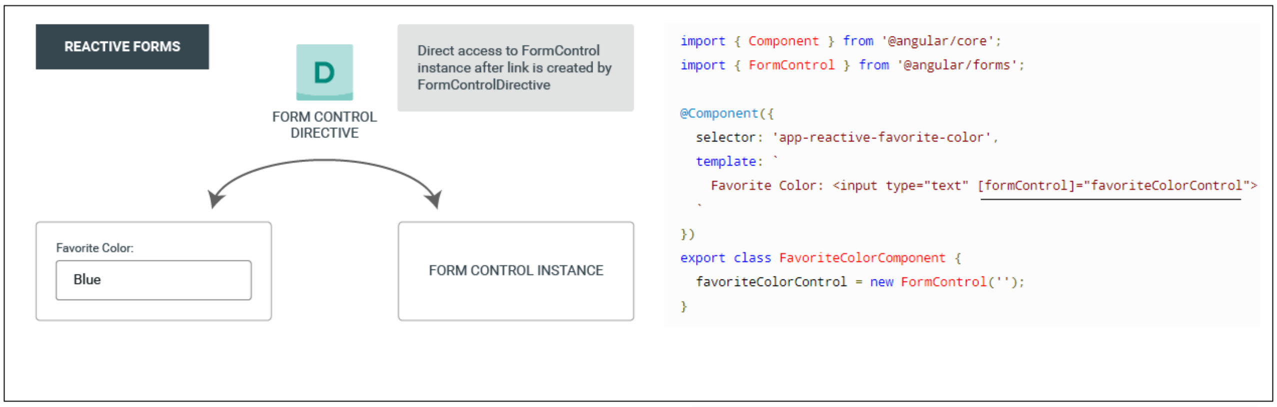
Reactive forms and template-driven forms process and manage form data differently. Each approach offers different advantages.

- **Reactive forms** provide direct, explicit access to the underlying forms object model. Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.
- **Template-driven forms** rely on directives in the template to create and manipulate the underlying object model.

Common form foundation classes

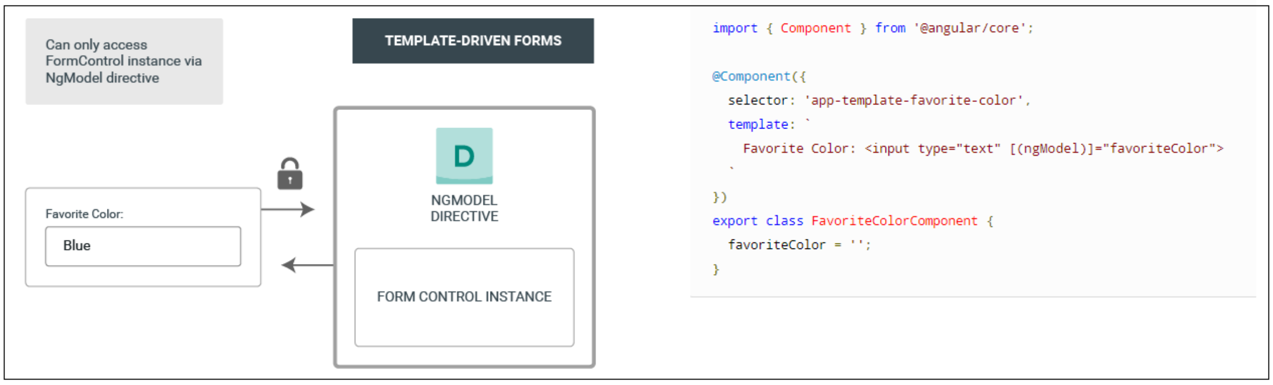
Both reactive and template-driven forms are built on the following base classes.

- **FormControl** tracks the value and validation status of an individual form control.
- **FormGroup** tracks the same values and status for a collection of form controls.
- **FormArray** tracks the same values and status for an array of form controls.
- **ControlValueAccessor** creates a bridge between Angular FormControl instances and native DOM elements.



REACTIVE FORMS
D
FORM CONTROL DIRECTIVE
Direct access to FormControl instance after link is created by FormControlDirective
Favorite Color:
Blue
FORM CONTROL INSTANCE

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text" [formControl]="favoriteColorControl">
  `
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```



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

@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `
})
export class FavoriteColorComponent {
  favoriteColor = '';
}
```

7) Angular Terminology and Definition

ahead-of-time (AOT) compilation

The Angular ahead-of-time (AOT) compiler converts Angular HTML and TypeScript code into efficient JavaScript code during the build phase, before the browser downloads and runs that code. This is the best compilation mode for production environments, with decreased load time and increased performance compared to just-in-time (JIT) compilation.

By compiling your application using the ngc command-line tool, you can bootstrap directly to a module factory, so you don't need to include the Angular compiler in your JavaScript bundle.

app-shell

App shell is a way to render a portion of your application via a route at build time. This gives users a meaningful first paint of your application that appears quickly because the browser can render static HTML and CSS without the need to initialize JavaScript.

You can use the Angular CLI to generate an app shell. This can improve the user experience by quickly launching a static rendered page (a skeleton common to all pages) while the browser downloads the full client version and switches to it automatically after the code loads.

Architect

The tool that the CLI uses to perform complex tasks such as compilation and test running, according to a provided configuration. Architect is a shell that runs a builder (defined in an npm package) with a given target configuration.In the workspace configuration file, an "architect" section provides configuration options for Architect builders.

Use the CLI command ng run to invoke a builder by specifying a target configuration associated with that builder. Integrators can add builders to enable tools and workflows to run through the Angular CLI. For example, a custom builder can replace the third-party tools used by the built-in implementations for CLI commands such as ng build or ng test.

attribute directives

A category of directive that can listen to and modify the behavior of other HTML elements, attributes, properties, and components. They are usually represented as HTML attributes, hence the name.

bootstrap

A way to initialize and launch an app or system.

In Angular, an app's root NgModule (AppModule) has a bootstrap property that identifies the app's top-level components. During the bootstrap process, Angular creates and inserts these components into the index.html host web page. You can bootstrap multiple apps in the same index.html. Each app contains its own components.

builder

A function that uses the Architect API to perform a complex process such as "build" or "test". The builder code is defined in an npm package.

For example, BrowserBuilder runs a webpack build for a browser target and KarmaBuilder starts the Karma server and runs a webpack build for unit tests.

The CLI command ng run invokes a builder with a specific target configuration. The workspace configuration file, angular.json, contains default configurations for built-in builders.

change detection

The mechanism by which the Angular framework synchronizes the state of an application's UI with the state of the data. The change detector checks the current state of the data model whenever it runs, and maintains it as the previous state to compare on the next iteration.

As the application logic updates component data, values that are bound to DOM properties in the view can change. The change detector is responsible for updating the view to reflect the current data model. Similarly, the user can interact with the UI, causing events that change the state of the data model. These events can trigger change detection.

component

A class with the @Component() decorator that associates it with a companion template. Together, the component class and template define a view. A component is a special type of directive.
The @Component() decorator extends the @Directive() decorator with template-oriented features.

An Angular component class is responsible for exposing data and handling most of the view's display and user-interaction logic through data binding.

data binding

A process that allows apps to display data values to a user and respond to user actions (such as clicks, touches, and keystrokes).

In data binding, you declare the relationship between an HTML widget and a data source and let the framework handle the details. Data binding is an alternative to manually pushing application data values into HTML, attaching event listeners, pulling changed values from the screen, and updating application data values.

Read about the following forms of binding in Angular's Template Syntax:

- Interpolation
- Property binding
- Event binding
- Attribute binding
- Class binding
- Style binding
- Two-way data binding with ngModel

declarable

A class type that you can add to the declarations list of an NgModule. You can declare components, directives, and pipes.

decorator | decoration

A function that modifies a class or property definition. Decorators (also called *annotations*) are an experimental (stage 2) JavaScript language feature. TypeScript adds support for decorators.

Angular defines decorators that attach metadata to classes or properties so that it knows what those classes or properties mean and how they should work.

Dependency injection

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

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

- In Angular, the DI framework provides declared dependencies to a class when that class is instantiated.

- Having multiple classes in the same file can be confusing. We generally recommend that you define components and services in separate files.
- If you do combine a component and service in the same file, it is important to define the service first, and then the component. If you define the component before the service, you get a run-time null reference error.

Configure an injector with a service provider

The class we have created provides a service. The @Injectable() decorator marks it as a service that can be injected, but Angular can't actually inject it anywhere until you configure an Angular dependency injector with a provider of that service.

The injector is responsible for creating service instances and injecting them into classes like HeroListComponent. You rarely create an Angular injector yourself. Angular creates injectors for you as it executes the app, starting with the *root injector* that it creates during the bootstrap process.

- Injectors are inherited, which means that if a given injector can't resolve a dependency, it asks the parent injector to resolve it. A component can get services from its own injector, from the injectors of its component ancestors, from the injector of its parent NgModule, or from the root injector.

You can configure injectors with providers at different levels of your app, by setting a metadata value in one of three places:
- In the @Injectable() decorator for the service itself.
- In the @NgModule() decorator for an NgModule.
- In the @Component() decorator for a component.

The @Injectable() decorator has the providedIn metadata option, where you can specify the provider of the decorated service class with the root injector, or with the injector for a specific NgModule.

The @NgModule() and @Component() decorators have the providers metadata option, where you can configure providers for NgModule-level or component-level injectors.

- Components are directives, and the providers option is inherited from @Directive(). You can also configure providers for directives and pipes at the same level as the component.

Injecting services

- You can tell Angular to inject a dependency in a component's constructor by specifying a constructor parameter with the dependency type.
- Service must be provided in some parent injector. The code in component doesn't depend on where Service comes from. If you decided to provide Service in AppModule, Component wouldn't change.

Injector hierarchy and service instances

Services are singletons *within the scope of an injector*. That is, there is at most one instance of a service in a given injector.

There is only one root injector for an app. Providing UserService at the root or AppModule level means it is registered with the root injector. There is just one UserService instance in the entire app and every class that injects UserService gets this service instance *unless* you configure another provider with a *child injector*.

Angular DI has a hierarchical injection system, which means that nested injectors can create their own service instances. Angular regularly creates nested injectors. Whenever Angular creates a new instance of a component that has providers specified in @Component(), it also creates a new *child injector* for that instance. Similarly, when a new NgModule is lazy-loaded at run time, Angular can create an injector for it with its own providers.

Child modules and component injectors are independent of each other, and create their own separate instances of the provided services. When Angular destroys an NgModule or component instance, it also destroys that injector and that injector's service instances.

Dependency injection tokens

When you configure an injector with a provider, you associate that provider with a DI token. The injector maintains an internal *token-provider* map that it references when asked for a dependency. The token is the key to the map.

Optional dependencies

When a component or service declares a dependency, the class constructor takes that dependency as a parameter. You can tell Angular that the dependency is optional by annotating the constructor parameter with @Optional().

```
constructor(@Optional() private logger?: Logger) {
  if (this.logger) {
    this.logger.log(someMessage);
  }
}
```

differential loading

A build technique that creates two bundles for an application. One smaller bundle is for modern browsers. A second, larger bundle allows the application to run correctly in older browsers (such as IE11) that do not support all modern browser APIs.

directive

A class that can modify the structure of the DOM or modify attributes in the DOM and component data model. A directive class definition is immediately preceded by a @Directive() decorator that supplies metadata.

A directive class is usually associated with an HTML element or attribute, and that element or attribute is often referred to as the directive itself. When Angular finds a directive in an HTML template, it creates the matching directive class instance and gives the instance control over that portion of the browser DOM.

There are three categories of directive:

- Components use @Component() (an extension of @Directive()) to associate a template with a class.
- Attribute directives modify behavior and appearance of page elements.
- Structural directives modify the structure of the DOM.

Angular supplies a number of built-in directives that begin with the ng prefix. You can also create new directives to implement your own functionality.

domain-specific language (DSL)

Angular extends TypeScript with domain-specific languages for a number of domains relevant to Angular apps, defined in NgModules such as animations, forms, and routing and navigation.

dynamic component loading

A technique for adding a component to the DOM at run time. Requires that you exclude the component from compilation and then connect it to Angular's change-detection and event-handling framework when you add it to the DOM.

eager loading

NgModules or components that are loaded on launch are called eager-loaded

element

Angular defines an ElementRef class to wrap render-specific native UI elements. In most cases, this allows you to use Angular templates and data binding to access DOM elements without reference to the native element.

entry point

A JavaScript module that is intended to be imported by a user of an npm package. An entry-point module typically re-exports symbols from other internal modules. A package can contain multiple entry points. For example, the @angular/core package has two entry-point modules, which can be imported using the module names @angular/core and @angular/core/testing.

form control

A instance of FormControl, which is a fundamental building block for Angular forms. Together with FormGroup and FormArray, tracks the value, validation, and status of a form input element.

form model

The "source of truth" for the value and validation status of a form input element at a given point in time. When using reactive forms, the form model is created explicitly in the component class. When using template-driven forms, the form model is implicitly created by directives.

immutability

The ability to alter the state of a value after its creation. Reactive forms perform immutable changes in that each change to the data model produces a new data model rather than modifying the existing one. Template-driven forms perform mutable changes with NgModel and two-way data binding to modify the existing data model in place.

injectable

An Angular class or other definition that provides a dependency using the dependency injection mechanism. An injectable service class must be marked by the @Injectable() decorator.

injector

An object in the Angular dependency-injection system that can find a named dependency in its cache or create a dependency using a configured provider. Injectors are created for NgModules automatically as part of the bootstrap process and are inherited through the component hierarchy.

- An injector provides a singleton instance of a dependency, and can inject this same instance in multiple components.
- A hierarchy of injectors at the NgModule and component level can provide different instances of a dependency to their own components and child components.
- You can configure injectors with different providers that can provide different implementations of the same dependency.

input

When defining a directive, the @Input() decorator on a directive property makes that property available as a *target* of a property binding.

just-in-time (JIT) compilation

The Angular just-in-time (JIT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code at run time, as part of bootstrapping.

JIT compilation is the default (as opposed to AOT compilation) when you run Angular's ng build and ng serve CLI commands, and is a good choice during development. JIT mode is strongly discouraged for production use because it results in large application payloads that hinder the bootstrap performance.

lazy loading

A process that speeds up application load time by splitting the application into multiple bundles and loading them on demand. For example, dependencies can be lazy loaded as needed—as opposed to eager-loaded modules that are required by the root module and are thus loaded on launch.

The router makes use of lazy loading to load child views only when the parent view is activated. Similarly, you can build custom elements that can be loaded into an Angular app when needed.

library

In Angular, a project that provides functionality that can be included in other Angular apps. A library isn't a complete Angular app and can't run independently. (To add re-usable Angular functionality to non-Angular web apps, you can use Angular custom elements.)

- Library developers can use the Angular CLI to generate scaffolding for a new library in an existing workspace, and can publish a library as an npm package.
- Application developers can use the Angular CLI to add a published library for use with an application in the same workspace.

lifecycle hook

An interface that allows you to tap into the lifecycle of directives and components as they are created, updated, and destroyed.

Each interface has a single hook method whose name is the interface name prefixed with ng. For example, the OnInit interface has a hook method named ngOnInit.

Angular calls these hook methods in the following order:

- ngOnChanges: When an input/output binding value changes.
- ngOnInit: After the first ngOnChanges.
- ngDoCheck: Developer's custom change detection.
- ngAfterContentInit: After component content initialized.
- ngAfterContentChecked: After every check of component content.
- ngAfterViewInit: After a component's views are initialized.
- ngAfterViewChecked: After every check of a component's views.
- ngOnDestroy: Just before the directive is destroyed.

ngcc

Angular compatibility compiler. If you build your app using Ivy, but it depends on libraries that have not been compiled with Ivy, the CLI uses ngcc to automatically update the dependent libraries to use Ivy.

observable

A producer of multiple values, which it pushes to subscribers. Used for asynchronous event handling throughout Angular. You execute an observable by subscribing to it with its subscribe() method, passing callbacks for notifications of new values, errors, or completion.

Observables can deliver single or multiple values of any type to subscribers, either synchronously (as a function delivers a value to its caller) or on a schedule. A subscriber receives notification of new values as they are produced and notification of either normal completion or error completion.

observer

An object passed to the subscribe() method for an observable.

output

When defining a directive, the @Output{} decorator on a directive property makes that property available as a *target* of event binding.

pipe

A class which is preceded by the @Pipe{} decorator and which defines a function that transforms input values to output values for display in a view. Angular defines various pipes, and you can define new pipes.

Polyfills

Angular is built on the latest standards of the web platform. Targeting such a wide range of browsers is challenging because they do not support all features of modern browsers. You compensate by loading polyfill scripts ("polyfills") for the browsers that you must support.

> The suggested polyfills are the ones that run full Angular applications. You may need additional polyfills to support features not covered by this list. Note that polyfills cannot magically transform an old, slow browser into a modern, fast one.

In Angular CLI version 8 and higher, applications are built using *differential loading*, a strategy where the CLI builds two separate bundles as part of your deployed application.

- The first bundle contains modern ES2015 syntax, takes advantage of built-in support in modern browsers, ships less polyfills, and results in a smaller bundle size.
- The second bundle contains code in the old ES5 syntax, along with all necessary polyfills. This results in a larger bundle size, but supports older browsers.

This strategy allows you to continue to build your web application to support multiple browsers, but only load the necessary code that the browser needs.

provider

An object that implements one of the Provider interfaces. A provider object defines how to obtain an injectable dependency associated with a DI token. An injector uses the provider to create a new instance of a dependency for a class that requires it.

Angular registers its own providers with every injector, for services that Angular defines. You can register your own providers for services that your app needs.

reactive forms

A framework for building Angular forms through code in a component. The alternative is a template-driven form.

When using reactive forms:

- The "source of truth", the form model, is defined in the component class.
- Validation is set up through validation functions rather than validation directives.
- Each control is explicitly created in the component class by creating a FormControl instance manually or with FormBuilder.
- The template input elements do *not* use ngModel.
- The associated Angular directives are prefixed with form, such as formControl, formGroup, and formControlName.

template-driven forms

A format for building Angular forms using HTML forms and input elements in the view. The alternative format uses the reactive forms framework.

When using template-driven forms:

- The "source of truth" is the template. The validation is defined using attributes on the individual input elements.
- Two-way binding with ngModel keeps the component model synchronized with the user's entry into the input elements.
- Behind the scenes, Angular creates a new control for each input element, provided you have set up a name attribute and two-way binding for each input.

- The associated Angular directives are prefixed with ng such as ngForm, ngModel, and ngModelGroup.

resolver

A class that implements the Resolve interface (or a function with the same signature as the resolve() method) that you use to produce or retrieve data that is needed before navigation to a requested route can be completed.

Resolvers run after all route guards for a route tree have been executed and have succeeded.

route guard

A method that controls navigation to a requested route in a routing application. Guards determine whether a route can be activated or deactivated, and whether a lazy-loaded module can be loaded.

router

A tool that configures and implements navigation among states and views within an Angular app.

The Router module is an NgModule that provides the necessary service providers and directives for navigating through application views. A routing component is one that imports the Router module and whose template contains a RouterOutlet element where it can display views produced by the router.

The router defines navigation among views on a single page, as opposed to navigation among pages. It interprets URL-like links to determine which views to create or destroy, and which components to load or unload. It allows you to take advantage of lazy loading in your Angular apps.

router outlet

A directive that acts as a placeholder in a routing component's template. Angular dynamically renders the template based on the current router state.

routing component

An Angular component with a RouterOutlet directive in its template that displays views based on router navigations.

rule

In schematics, a function that operates on a file tree to create, delete, or modify files in a specific manner.

schematic

A scaffolding library that defines how to generate or transform a programming project by creating, modifying, refactoring, or moving files and code. A schematic defines rules that operate on a virtual file system called a tree.

The Angular CLI uses schematics to generate and modify Angular projects and parts of projects.
- Angular provides a set of schematics for use with the CLI. See the Angular CLI command reference. The ng add command runs schematics as part of adding a library to your project. The ng generate command runs schematics to create apps, libraries, and Angular code constructs.
- Library developers can create schematics that enable the Angular CLI to add and update their published libraries, and to generate artifacts the library defines. Add these schematics to the npm package that you use to publish and share your library.

scoped package

A way to group related npm packages. NgModules are delivered within scoped packages whose names begin with the Angular *scope name* @angular. For example, @angular/core, @angular/common, @angular/forms, and @angular/router.

server-side rendering

A technique that generates static application pages on the server, and can generate and serve those pages in response to requests from browsers. It can also pre-generate pages as HTML files that you serve later.

This technique can improve performance on mobile and low-powered devices and improve the user experience by showing a static first page quickly while the client-side app is loading. The static version can also make your app more visible to web crawlers.

You can easily prepare an app for server-side rendering by using the CLI to run the Angular Universal tool, using the @nguniversal/express-engine schematic.

service

In Angular, a class with the @Injectable() decorator that encapsulates non-UI logic and code that can be reused across an application. Angular distinguishes components from services to increase modularity and reusability.

The @Injectable() metadata allows the service class to be used with the dependency injection mechanism. The injectable class is instantiated by a provider. Injectors maintain lists of providers and use them to provide service instances when they are required by components or other services.

structural directives

A category of directive that is responsible for shaping HTML layout by modifying the DOM&mdashthat is, adding, removing, or manipulating elements and their children.

subscriber

A function that defines how to obtain or generate values or messages to be published. This function is executed when a consumer calls the subscribe() method of an observable.

The act of subscribing to an observable triggers its execution, associates callbacks with it, and creates a Subscription object that lets you unsubscribe.

The subscribe() method takes a JavaScript object (called an observer) with up to three callbacks, one for each type of notification that an observable can deliver:

- The next notification sends a value such as a number, a string, or an object.
- The error notification sends a JavaScript Error or exception.
- The complete notification doesn't send a value, but the handler is called when the call completes. Scheduled values can continue to be returned after the call completes.

target

A buildable or runnable subset of a project, configured as an object in the workspace configuration file, and executed by an Architect builder.

In the angular.json file, each project has an "architect" section that contains targets which configure builders. Some of these targets correspond to CLI commands, such as build, serve, test, and lint.

For example, the Architect builder invoked by the ng build command to compile a project uses a particular build tool, and has a default configuration whose values can be overridden on the command line.
The build target also defines an alternate configuration for a "production" build, that can be invoked with the --prod flag on the build command.

The Architect tool provides a set of builders. The ng new command provides a set of targets for the initial application project. The ng generate application and ng generate library commands provide a set of targets for each new project. These targets, their options and configurations, can be customized to meet the needs of your project. For example, you may want to add a "staging" or "testing" configuration to a project's "build" target.

template

Code that defines how to render a component's view.

A template combines straight HTML with Angular data-binding syntax, directives, and template expressions (logical constructs). The Angular elements insert or calculate values that modify the HTML elements before the page is displayed. Learn more about Angular template language in the Template Syntax guide.

A template is associated with a component class through the @Component() decorator. The template code can be provided inline, as the value of the template property, or in a separate HTML file linked through the templateUrl property.

template reference variable

A variable defined in a template that references an instance associated with an element, such as a directive instance, component instance, template as in TemplateRef, or DOM element. After declaring a template reference variable on an element in a template, you can access values from that variable elsewhere within the same template. The following example defines a template reference variable named #phone.

```
<input #phone placeholder="phone number" />

<button (click)="callPhone(phone.value)">Call</button>

-----------------------------
<form #itemForm="ngForm" (ngSubmit)="onSubmit(itemForm)">
<input class="form-control" name="name" ngModel required />
 <button type="submit">Submit</button>
 </form>
<div [hidden]="!itemForm.form.valid"> <p>{{ submitMessage }}</p> </div>
```

token

An opaque identifier used for efficient table lookup. In Angular, a DI token is used to find providers of dependencies in the dependency injection system.

transpile

The translation process that transforms one version of JavaScript to another version; for example, down-leveling ES2015 to the older ES5 version.

tree

In schematics, a virtual file system represented by the Tree class. Schematic rules take a tree object as input, operate on them, and return a new tree object.

TypeScript

A programming language based on JavaScript that is notable for its optional typing system. TypeScript provides compile-time type checking and strong tooling support (such as code completion, refactoring, inline documentation, and intelligent search). Many code editors and IDEs support TypeScript either natively or with plug-ins.

TypeScript configuration file

A file specifies the root files and the compiler options required to compile a TypeScript project.

TypeScript is a primary language for Angular application development. It is a superset of JavaScript with design-time support for type safety and tooling.

Browsers can't execute TypeScript directly. Typescript must be "transpiled" into JavaScript using the *tsc* compiler, which requires some configuration.

A given Angular workspace contains several TypeScript configuration files. At the root level, there are two main TypeScript configuration files: a tsconfig.json file and a tsconfig.base.json file.

The tsconfig.json file is a "Solution Style" TypeScript configuration file. Code editors and TypeScript's language server use this file to improve development experience. Compilers do not use this file.

SOLUTION STYLE (WE CAN CONFIGURE OTHER TSCONFIG FILES AS PER REQUIREMENT)

Editors need to figure out which configuration file a file belongs to so that it can apply the appropriate options and figure out which other files are included in the current "project". By default, editors powered by TypeScript's language server do this by walking up each parent directory to find a tsconfig.json.

```
{
   "files": [],
   "references": [
      { "path": "./tsconfig.shared.json" },
      { "path": "./tsconfig.frontend.json" },
      { "path": "./tsconfig.backend.json" },
   ]
}
```

unidirectional data flow

A data flow model where the component tree is always checked for changes in one direction (parent to child), which prevents cycles in the change detection graph.

In practice, this means that data in Angular flows downward during change detection. A parent component can easily change values in its child components because the parent is checked first. A failure could occur, however, if a child component tries to change a value in its parent during change detection (inverting the expected data flow), because the parent component has already been rendered. In development mode, Angular throws the ExpressionChangedAfterItHasBeenCheckedError error if your app attempts to do this, rather than silently failing to render the new value.

To avoid this error, a lifecycle hook method that seeks to make such a change should trigger a new change detection run.

Universal

A tool for implementing server-side rendering of an Angular application. When integrated with an app, Universal generates and serves static pages on the server in response to requests from browsers. The initial static page serves as a fast-loading placeholder while the full application is being prepared for normal execution in the browser.

view

The smallest grouping of display elements that can be created and destroyed together. Angular renders a view under the control of one or more directives.

A component class and its associated template define a view. A view is specifically represented by a ViewRef instance associated with a component. A view that belongs immediately to a component is called a *host view*. Views are typically collected into view hierarchies.

Properties of elements in a view can change dynamically, in response to user actions; the structure (number and order) of elements in a view can't. You can change the structure of elements by inserting, moving, or removing nested views within their view containers.

View hierarchies can be loaded and unloaded dynamically as the user navigates through the application, typically under the control of a router.

View Engine

The compilation and rendering pipeline used by Angular before version 9.

view hierarchy

A tree of related views that can be acted on as a unit. The root view is a component's *host view*. A host view can be the root of a tree of *embedded views*, collected in a *view container* (ViewContainerRef) attached to an anchor element in the hosting component. The view hierarchy is a key part of Angular change detection.

The view hierarchy doesn't imply a component hierarchy. Views that are embedded in the context of a particular hierarchy can be host views of other components. Those components can be in the same NgModule as the hosting component, or belong to other NgModules.

workspace

A collection of Angular projects (that is, applications and libraries) powered by the Angular CLI that are typically co-located in a single source-control repository

The CLI ng new command creates a file system directory (the "workspace root"). In the workspace root, it also creates the workspace configuration file (angular.json) and, by default, an initial application project with the same name.

Commands that create or operate on apps and libraries (such as add and generate) must be executed from within a workspace folder.

workspace configuration

A file named angular.json at the root level of an Angular workspace provides workspace-wide and project-specific configuration defaults for build and development tools that are provided by or integrated with the Angular CLI.

Additional project-specific configuration files are used by tools, such as package.json for the npm package manager, tsconfig.json for TypeScript transpilation, and tslint.json for TSLint

zone

An execution context for a set of asynchronous tasks. Useful for debugging, profiling, and testing apps that include asynchronous operations such as event processing, promises, and calls to remote servers.

An Angular app runs in a zone where it can respond to asynchronous events by checking for data changes and updating the information it displays by resolving data bindings.

8) RXJS

RxJS is a library for composing asynchronous and event-based programs by using observable sequences. It contains following major key contains:-

1.Operators
3.subject=>An RxJS Subject is a special type of Observable that allows values to be multi-casted to many Observable.

- **1.Subject** - No initial value or replay behavior.
- **2.AsyncSubject** - Emits latest value to observers upon completion.
- **3.BehaviorSubject** - Requires an initial value and emits its current value (last emitted item) to new subscribers.
- **4.ReplaySubject** - Emits specified number of last emitted values (a replay) to new subscribers.

3. Observalble and subscriber

Observable=>Observables are lazy Push collections of multiple values.
subscriber=> A Subscription is an object that represents a disposable resource, usually the execution of an Observable.

Q. why Observable ?
➔general method, return only one value but observable can return multiple values;

```
function foo() {
  console.log('Hello');
  return 42;
}
const x = foo.call(); // same as foo()
console.log(x);
```

1.Operators

➔ operators -> perform operations on observable

1.creational operators : These operator will return observable.
2.pipable operators :

➔A Pipeable Operator is a function that takes an Observable as its input and returns another Observable. It is a pure operation: the previous Observable stays unmodified.

➔You can use pipes to link operators together. Pipes let you combine multiple functions into a single function.
➔The pipe() function takes as its arguments the functions you want to combine, and returns a new function that, when executed, runs the composed functions in sequence.

(a)single operator: perform one operation at a time	**(b)multiple operation with pipe()**

const nums = **of**(1, 2, 3);
const squareValues = **map**((val: number) => val * val);
const squaredNums = **squareValues**(nums);
squaredNums.subscribe(x => console.log(x));
or
const m=[1, 2, 3].**map**(x => x * x);
m.subscribe(x=>console.log(x));
or
map(x => x * x)(of(1, 2, 3)).subscribe((v) => console.log(\`value: ${v}\`));
or
	const nums = of(1, 2, 3, 4, 5);
	const squareOddVals = pipe(
	filter((n: number) => n % 2 !== 0),
	map(n => n * n)
	);
	const squareOdd = squareOddVals(nums);
	squareOdd.subscribe(x => console.log(x));

```
first()(of(1, 2, 3)).subscribe((v) => console.log(`value: ${v}`));
//output value 1
```

(c)pipe() is also an function of observable ,we can directly use it with observable

```
const squareOdd = of(1, 2, 3, 4, 5)
                        .pipe(
                                filter(n => n % 2 !== 0),
                                map(n => n * n)
                        );
squareOdd.subscribe(x => console.log(x));
```

(d) error handling with operators

```
const apiData = ajax('/api/data').pipe(
            retry(3),
            map(res => {
             if (!res.response) {
               throw new Error('Value expected!');
             }
             return res.response;
               }), catchError(err => of([])));

apiData.subscribe({
next(x) { console.log('data: ', x); },
error(err) { console.log('errors '); }});
```

2. SUBJECT

1.subject

```
            const sub = new Subject();

            sub.next(1);
            sub.subscribe(x => {
              console.log('Subscriber A', x)  // may be 1,2,4,3
            });
            sub.next(2);
            sub.subscribe(x => {
              console.log('Subscriber B', x)
            });
            sub.next(4);
            sub.next(3);

            Output:

            Subscriber A 2
            Subscriber A 4
            Subscriber B 4
            Subscriber A 3
            Subscriber B 3
```

note:-
you will never get seqencial result.
if value will change ,it will call all subscriber again with updated value.

2.BehaviorSubject

```
const subject = new BehaviorSubject(123);
subject.subscribe(console.log); // 123

subject.next(456);
subject.subscribe(console.log); //456

subject.next(789);

// output: 123, [456, 456], [789, 789]
```

note:-
you will get sequncial result as per the call;
if value will change ,it will call all subscriber again with updated value.

3.ReplaySubject

```
const sub = new ReplaySubject(3);

sub.next(1);
sub.next(2);
sub.next(3);
sub.next(4);
sub.subscribe(x=>console.log("a"+x));
sub.next(5);
sub.next(6);
sub.subscribe(x=>console.log("b"+x));
sub.next(7);

output:-[a2,a3,a4],[a5],[a6],[b4,b5,b6],[a7,b7]
```

note:-
first subscriber will call last three updated value,
and if we add new value emediatly it will emit current value to all above subscriber.

4.AsyncSubject

```
            const sub = new AsyncSubject();

            sub.subscribe(console.log);

            sub.next(123); //nothing logged

            sub.subscribe(console.log);

            sub.next(456); //nothing logged
            sub.complete();

//==> 456, 456 logged by both subscribers
```

note:-
untill we call complete method ,it will not emit value to any subscriber and after complete method call ,it will emit only latest value to all subscriber

3. Observable and observer

Area	OPERATORS
1. Creation	interval,of,from,fromEvent,ajax,bindCallback,bindNodeCallback,defer,empty,fromEventPattern,generate,range,throwError,timer,iif
2. join Creation	combineLatest,concat , forkJoin,merge,race,zip, [using pipe=>combineAll,concatAll,exhaust,mergeAll,startWith,withLatestFrom]
3. Transformation	buffer,bufferCount,bufferTime,bufferToggle,bufferWhen,concatMap,concatMapTo,exhaust,exhaustMap,expand,groupBy,map,mapTo,ergeMap,mergeMapTo,mergeScan,pairwise,partition,pluck,scan,switchMap,switchMapTo,window,windowCount,windowTime,windowToggle,windowWhen
4. Filtering	audit,,auditTime,debounce,debounceTime,distinct,distinctKey,distinctUntilChanged,distinctUntilKeyChanged,elementAt,filter,first,ignoreElements,last,sample,sampleTime,single,skip,skipLast,skipUntil,skipWhile,take,takeLast,takeUntil,takeWhile,throttle,throttleTime
5. Multicasting	multicast,publish,publishLast
6. Error Handling	catchError,retry,retryWhen
7. Utility	tap,delay,delayWhen,dematerialize,materialize,observeOn,subscribeOn,timeInterval,timestamp,timeout,timeoutWith,toArray
8. Conditional and Boolean	defaultIfEmpty,every,find,findIndex,isEmpty
9. Mathematical and Aggregate	count,max,min,reduce

1.Creational Operator	

interval	// Creates an Observable that emits sequential numbers every specified interval of time, on a specified SchedulerLike.
	
	const numbers = interval(1000);
	numbers.subscribe(x => console.log(x));

of	// Converts the arguments to an observable sequence.
	
	const numbers = of(1,2,3,4);
	numbers.subscribe(x => console.log(x));

from	//Creates an Observable from an Array, an array-like object, a Promise, an iterable object, or an Observable-like object.
	
	const numbers = from([1,2,3,4]);
	numbers.subscribe(x => console.log(x));

fromEvent	//Creates an Observable that emits events of a specific type coming from the given event target.
	
	const clicks = fromEvent(document, 'click');
	clicks.subscribe(x => console.log(x));

ajax	const obs$ = ajax(`https://api.github.com/users?per_page=5`)
	.pipe(
	map(userResponse => console.log('users: ', userResponse)),
	catchError(error => {
	console.log('error: ', error);
	return of(error); })
	);
	obs$.subscribe(x=>console.log(x));

bindCallback	
bindNodeCallback	

defer	// Creates the Observable lazily, that is, only when it is subscribed. untill u subscribe it ,it will not instantiate observable
	
	const clicksOrInterval = defer(function () { return interval(1000);});
	clicksOrInterval.subscribe(x => console.log(x));

empty	const interval$ = interval(1000);
	const result = interval$.pipe(mergeMap(x => x % 2 === 1 ? of('a', 'b', 'c') : empty()));
	result.subscribe(x => console.log(x));

fromEventPattern	// Creates an Observable from an arbitrary API for registering event handlers.
	
	function addClickHandler(handler) {
	document.addEventListener('click', handler);
	}

	function removeClickHandler(handler) {
	document.removeEventListener('click', handler);
	}
	
	const clicks = fromEventPattern(addClickHandler , removeClickHandler);
	clicks.subscribe(x => console.log(x));

generate	

range	//Creates an Observable that emits a sequence of numbers within a specified range.
	
	const numbers = range(1, 10);
	numbers.subscribe(x => console.log(x));

throwError	// Creates an Observable that emits no items to the Observer and immediately emits an error notification.
	
	interval(1000).pipe(mergeMap(x => x === 2? throwError('Twos are bad'): of('a', 'b', 'c'))
	.subscribe(x => console.log(x), e => console.error(e));

timer	//Creates an Observable that starts emitting after an dueTime and emits ever increasing numbers after each period of time thereafter.
	
	const numbers = timer(3000, 1000);

iif	// Change at runtime which Observable will be subscribed
	
	let subscribeToFirst;
	const firstOrSecond = iif(() => subscribeToFirst, of('first'), of('second'));
	
	subscribeToFirst = true;
	firstOrSecond.subscribe(value => console.log(value)); // "first"
	
	subscribeToFirst = false;
	firstOrSecond.subscribe(value => console.log(value)); // "second"

2. join Creation	

combineLatest	// Combines multiple Observables to create an Observable whose values are calculated from the latest values of each of its input Observables.
	
	const firstTimer = timer(0, 1000); // emit 0, 1, 2... after every second, starting from now
	const secondTimer = timer(500, 1000); // emit 0, 1, 2... after every second, starting 0,5
	combinedTimers = combineLatest(firstTimer, secondTimer);
	combinedTimers.subscribe(value => console.log(value));

concat	// Creates an output Observable which sequentially emits all values from given Observable and then moves on to the next.
	
	const result = concat(firstTimer, secondTimer);
	result.subscribe(x => console.log(x));

forkJoin	// Accepts an Array of ObservableInput or a dictionary Object of ObservableInput and returns
	// an Observable that emits either an array of values in the exact same order as the passed array,
	//or a dictionary of values in the same shape as the passed dictionary.
	
	const observable = forkJoin({
	foo: of(1, 2, 3, 4),
	bar: Promise.resolve(8),
	baz: timer(4000),
	});
	observable.subscribe({
	next: value => console.log(value),
	complete: () => console.log('This is how it ends!'),
	});
	
	// Logs:
	// { foo: 4, bar: 8, baz: 0 } after 4 seconds
	// "This is how it ends!" immediately after

merge	// Creates an output Observable which concurrently emits all values from every given input Observable.

	const clicks = fromEvent(document, 'click');
	const timer = interval(1000);
	const clicksOrTimer = merge(clicks, timer);
	clicksOrTimer.subscribe(x => console.log(x));

race	// Subscribes to the observable that was the first to start emitting.
	
	const obs1 = interval(1000).pipe(mapTo('fast one'));
	const obs2 = interval(3000).pipe(mapTo('medium one'));
	const obs3 = interval(5000).pipe(mapTo('slow one'));
	
	race(obs3, obs1, obs2).subscribe(winner => console.log(winner));

zip	//Combines multiple Observables to create an Observable whose values are calculated from the values,
	//in order, of each of its input Observables.
	
	let age$ = of<number>(27, 25, 29);
	let name$ = of<string>('Foo', 'Bar', 'Beer');
	let isDev$ = of<boolean>(true, true, false);
	
	zip(age$, name$, isDev$).pipe(map(([age, name, isDev]) => ({ age, name, isDev })))
	                                    .subscribe(x => console.log(x));
	
	// outputs
	// { age: 27, name: 'Foo', isDev: true }
	// { age: 25, name: 'Bar', isDev: true }
	// { age: 29, name: 'Beer', isDev: false }

Join Operation	

combineAll	// Flattens an Observable-of-Observables by applying combineLatest when the Observable-of-Observables completes.
	// Map two click events to a finite interval Observable, then apply combineAll
	
	const clicks = fromEvent(document, 'click');
	const higherOrder = clicks.pipe(
	                                        map(ev =>interval(Math.random() * 2000).pipe(take(3))),
	                                        take(2));
	const result = higherOrder.pipe(combineAll());
	result.subscribe(x => console.log(x));

concatAll	//Converts a higher-order Observable into a first-order Observable by concatenating the inner Observables in order.
	//For each click event, tick every second from 0 to 3, with no concurrency
	
	const clicks = fromEvent(document, 'click');
	const higherOrder = clicks.pipe(map(ev => interval(1000).pipe(take(4))));
	const firstOrder = higherOrder.pipe(concatAll());
	firstOrder.subscribe(x => console.log(x));

exhaust	// Converts a higher-order Observable into a first-order Observable by dropping inner Observables
	// while the previous inner Observable has not yet completed.
	//Run a finite timer for each click, only if there is no currently active timer
	
	const clicks = fromEvent(document, 'click');
	const higherOrder = clicks.pipe(map((ev) => interval(1000).pipe(take(5))));
	const result = higherOrder.pipe(exhaust());
	result.subscribe(x => console.log(x));

mergeAll	//Converts a higher-order Observable into a first-order Observable which concurrently delivers all values that are emitted on the inner Observables.
	//Count from 0 to 9 every second for each click, but only allow 2 concurrent timers
	
	const clicks = fromEvent(document, 'click');
	const higherOrder = clicks.pipe(map((ev) => interval(1000).pipe(take(10))));
	const firstOrder = higherOrder.pipe(mergeAll(2));
	firstOrder.subscribe(x => console.log(x));

startWith	// Returns an Observable that emits the items you specify as arguments before it begins to emit items emitted by the source Observable.
	// Start the chain of emissions with "first", "second"
	
	of("from source").pipe(startWith("first", "second")).subscribe(x => console.log(x));

withLatestFrom	// Combines the source Observable with other Observables to create an Observable whose values are calculated from the latest values of each, only when the source emits.

	//On every click event, emit an array with the latest timer event plus the click event
	
	const clicks = fromEvent(document, 'click');
	const timer = interval(1000);
	const result = clicks.pipe(withLatestFrom(timer));
	result.subscribe(x => console.log(x));

3. Transformation Operators	

buffer	// Buffers the source Observable values until closingNotifier emits.
	
	const clicks = fromEvent(document, 'click');
	const intervalEvents = interval(1000);
	const buffered = intervalEvents.pipe(buffer(clicks));
	
	 // clicking after 10s ,10 values will be bufferd at a time
	buffered.subscribe(x => console.log(x));

bufferCount	//Buffers the source Observable values until the size hits the maximum bufferSize given.
	// Emit the last two click events as an array
	
	const clicks = fromEvent(document, 'click');
	const buffered = clicks.pipe(bufferCount(2));
	buffered.subscribe(x => console.log(x));

bufferTime	// Buffers the source Observable values for a specific time period.
	//Every second, emit an array of the recent click events
	
	const clicks = fromEvent(document, 'click');
	const buffered = clicks.pipe(bufferTime(1000));
	buffered.subscribe(x => console.log(x));

bufferToggle	

bufferWhen	// Buffers the source Observable values, using a factory function of closing Observables
	// to determine when to close, emit, and reset the buffer.
	//Emit an array of the last clicks every [1-5] random seconds
	
	const clicks = fromEvent(document, 'click');
	const buffered = clicks.pipe(bufferWhen(() =>interval(1000 + Math.random() * 4000)));
	buffered.subscribe(x => console.log(x));

concatMap	// Projects each source value to an Observable which is merged in the output Observable,
	// in a serialized fashion waiting for each one to complete before merging the next.
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(
	concatMap(ev => interval(1000).pipe(take(4))));
	result.subscribe(x => console.log(x));

concatMapTo	// It's like concatMap, but maps each value always to the same inner Observable

exhaust	// Converts a higher-order Observable into a first-order Observable by dropping
	//inner Observables while the previous inner Observable has not yet completed.
	
	const clicks = fromEvent(document, 'click');
	const higherOrder = clicks.pipe(
	map((ev) => interval(1000).pipe(take(5))),
	);
	const result = higherOrder.pipe(exhaust());
	result.subscribe(x => console.log(x));

exhaustMap	// Projects each source value to an Observable which is merged in the output
	// Observable only if the previous projected Observable has completed.
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(
	exhaustMap(ev => interval(1000).pipe(take(5))));
	result.subscribe(x => console.log(x));

expand	//Recursively projects each source value to an Observable which is merged in the output Observable.
	
	const clicks = fromEvent(document, 'click');
	const powersOfTwo = clicks.pipe(
	mapTo(2),
	expand(x => of(2 * x).pipe(delay(1000))),
	take(10),
	);

	powersOfTwo.subscribe(x => console.log(x));

groupBy	// Groups the items emitted by an Observable according to a
	// specified criterion, and emits these grouped items as GroupedObservables, one GroupedObservable per group.
	
	of(
	    {id: 1, name: 'JavaScript'},
	    {id: 2, name: 'Parcel'},
	    {id: 2, name: 'webpack'},
	    {id: 1, name: 'TypeScript'},
	    {id: 3, name: 'TSLint'}
	).pipe(
	        groupBy(p => p.id),
	        mergeMap((group$) => group$.pipe(reduce((acc, cur) => [...acc, cur],
	        [])))
	 .subscribe(p => console.log(p));
	// displays:
	// [{ id: 1, name: 'JavaScript'},
	// { id: 1, name: 'TypeScript'}]
	//
	// [{ id: 2, name: 'Parcel'},
	// { id: 2, name: 'webpack'}]
	//
	// [{ id: 3, name: 'TSLint'}]

map	// Applies a given project function to each value emitted by the source Observable, and emits the resulting values as an Observable.
	
	const clicks = fromEvent(document, 'click');
	const positions = clicks.pipe(map(ev => ev.clientX));
	positions.subscribe(x => console.log(x));

mapTo	// Emits the given constant value on the output Observable every time the source Observable emits a value
	
	const clicks = fromEvent(document, 'click');
	const greetings = clicks.pipe(mapTo('Hi'));
	greetings.subscribe(x => console.log(x));

mergeMap	// Projects each source value to an Observable which is merged in the output Observable.
	
	const letters = of('a', 'b', 'c').pipe(mergeMap(x => interval(1000).pipe(map(i => x+i))));
	result.subscribe(x => console.log(x));

mergeMapTo	//Projects each source value to the same Observable which is merged multiple times in the output Observable.
	
	const clicks = fromEvent(document, 'click');
	const greetings = clicks.pipe(mapTo('Hi'));
	greetings.subscribe(x => console.log(x));

scan	// Applies an accumulator function over the source Observable, and returns each intermediate result, with an optional seed value.
	// Count the number of click events
	
	const clicks = fromEvent(document, 'click');
	const ones = clicks.pipe(mapTo(1));
	const seed = 0;
	const count = ones.pipe(scan((acc, one) => acc + one, seed));
	count.subscribe(x => console.log(x));

mergeScan	// Applies an accumulator function over the source Observable where the accumulator
	// function itself returns an Observable, then each intermediate Observable
	// returned is merged into the output Observable.
	
	const click$ = fromEvent(document, 'click');
	const one$ = click$.pipe(mapTo(1));
	const seed = 5;
	const count$ = one$.pipe(
	                mergeScan((acc, one) =>of(acc+one),seed),
	              );
	count$.subscribe(x => console.log(x)); //acc=seed(5),one=1 =>next acc=5+1,acc=6+1

Pairwise	//Groups pairs of consecutive emissions together and emits them as an array of two values.
partition	//Splits the source Observable into two, one with values that satisfy a predicate, and another with values that don't satisfy the predicate.

pluck	// Maps each source value (an object) to its specified nested property.
	// Map every click to the tagName of the clicked target element
	
	const clicks = fromEvent(document, 'click');
	const tagNames = clicks.pipe(pluck('target', 'tagName'));
	tagNames.subscribe(x => console.log(x));

switchMap	//Projects each source value to an Observable which is merged in the output Observable,
	//emitting values only from the most recently projected Observable.
	// Generate new Observable according to source Observable values
	
	//=>1,1^2,1^3..2,2^2,2^3
	const switched = of(1, 2, 3).pipe(
	        switchMap((x: number) => of(x, x ** 2, x ** 3)));
	switched.subscribe(x => console.log(x));

switchMapTo	// Projects each source value to the same Observable which is flattened multiple times with switchMap in the output Observable. Rerun an interval Observable on every click event
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(switchMapTo(interval(1000)));
	result.subscribe(x => console.log(x));

window	// In every window of 1 second each, emit at most 2 click events
	
	const clicks = fromEvent(document, 'click');
	const sec = interval(1000);
	const result = clicks.pipe(
	        window(sec),
	        map(win => win.pipe(take(2))),
	        mergeAll()
	        );
	result.subscribe(x => console.log(x));

windowCount	// Ignore every 3rd click event, starting from the first one
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(
	        windowCount(3),
	        map(win => win.pipe(skip(1))),
	        mergeAll()
	        );
	result.subscribe(x => console.log(x));

windowTime	//It's like bufferTime, but emits a nested Observable instead of an array.
	// In every window of 1 second each, emit at most 2 click events
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(
	        windowTime(1000),
	        map(win => win.pipe(take(2))),
	        mergeAll(),
	        );
	result.subscribe(x => console.log(x));

windowToggle	// Every other second, emit the click events from the next 500ms
	
	const clicks = fromEvent(document, 'click');
	const openings = interval(1000);
	const result = clicks.pipe(
	        windowToggle(openings, i => i % 2 ? interval(500) : EMPTY),
	        mergeAll()
	        );
	result.subscribe(x => console.log(x));

windowWhen	// Emit only the first two clicks events in every window of [1-5] random seconds
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(
	        windowWhen(() => interval(1000 + Math.random() * 4000)),
	        map(win => win.pipe(take(2))),
	        mergeAll());
	result.subscribe(x => console.log(x));

4.Filtering Operators	

audit	// Ignores source values for a duration determined by another Observable,
	// then emits the most recent value from the source Observable, then repeats this process.
	// Emit clicks at a rate of at most one click per second
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(audit(ev => interval(1000)));
	result.subscribe(x => console.log(x));

auditTime	//Ignores source values for duration milliseconds, then emits the most recent value from the source Observable,
	// then repeats this process.
	// Emit clicks at a rate of at most one click per second
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(auditTime(1000));
	result.subscribe(x => console.log(x));

debounce	// Emits a value from the source Observable only after a particular time span determined by another Observable
	// has passed without another source emission.
	// Emit the most recent click after a burst of clicks
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(debounce(() => interval(1000)));
	result.subscribe(x => console.log(x));

debounceTime	// Emits a value from the source Observable only after a particular time span has passed without another source emission.
	// Emit the most recent click after a burst of clicks
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(debounceTime(1000));
	result.subscribe(x => console.log(x));

distinct	// Returns an Observable that emits all items emitted by the source Observable that
	//are distinct by comparison from previous items.
	
	of(1, 1, 2, 2, 2, 1, 2, 3, 4, 3, 2, 1).pipe(distinct()).subscribe(x => console.log(x));
	
	// 1, 2, 3, 4

distinctKey	

distinctUntilChanged	// Returns an Observable that emits all items emitted by the
	// source Observable that are distinct by comparison from the previous item.
	of(1, 1, 2, 2, 2, 1, 1, 2, 3, 3, 4).pipe(distinctUntilChanged()).subscribe(x => console.log(x));
	
	// 1, 2, 1, 2, 3, 4

distinctUntilKeyChanged	// Returns an Observable that emits all items emitted by the source Observable that are distinct by c
	// omparison from the previous item,
	// using a property accessed by using the key provided to check if the two items are distinct.
	// An example comparing the name of persons
	
	interface Person {
	        age: number,
	        name: string
	}
	
	of<Person>(
	        { age: 4, name: 'Foo'},
	        { age: 7, name: 'Bar'},
	        { age: 5, name: 'Foo'},
	        { age: 6, name: 'Foo'},
	  ).pipe(distinctUntilKeyChanged('name')).subscribe(x => console.log(x));
	
	// displays:
	// { age: 4, name: 'Foo' }
	// { age: 7, name: 'Bar' }
	// { age: 5, name: 'Foo' }

elementAt	// Emits the single value at the specified index in a sequence of emissions from the source Observable.
	// Emit only the third click event
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(elementAt(2));
	result.subscribe(x => console.log(x));
	
	// Results in:
	// click 1 = nothing
	// click 2 = nothing
	// click 3 = MouseEvent object logged to console

filter	// Filter items emitted by the source Observable by only emitting those that satisfy a specified predicate.
	// Emit only click events whose target was a DIV element
	
	const clicks = fromEvent(document, 'click');
	const clicksOnDivs = clicks.pipe(filter(ev => ev.target.tagName === 'DIV'));
	clicksOnDivs.subscribe(x => console.log(x));

first	// Emits only the first value (or the first value that meets some condition) emitted by the source Observable.
	// Emit only the first click that happens on the DOM
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(first());
	result.subscribe(x => console.log(x));

ignoreElements	// Ignores all items emitted by the source Observable and only passes calls of complete or error.
	// Ignores emitted values, reacts to observable's completion.
	
	of('you', 'talking', 'to', 'me').pipe(ignoreElements())
	        .subscribe(
	          word => console.log(word),
	          err => console.log('error:', err),
	          () => console.log('the end'),
	        );
	// result:
	// 'the end'

last	

sample	//Emits the most recently emitted value from the source Observable whenever another Observable, the notifier, emits.
	// On every click, sample the most recent "seconds" timer
	
	const seconds = interval(1000);
	const clicks = fromEvent(document, 'click');
	const result = seconds.pipe(sample(clicks));
	result.subscribe(x => console.log(x));

sampleTime	// Emits the most recently emitted value from the source Observable within periodic time intervals.
	// Every second, emit the most recent click at most once
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(sampleTime(1000));
	result.subscribe(x => console.log(x));

single	// If the source Observable emits items but none match the specified predicate then undefined is emitted.
	
	const numbers = range(1,5).pipe(single(x => x === 10));
	numbers.subscribe(x => console.log(x)); // undefined

skip	// Returns an Observable that skips the first count items emitted by the source Observable.
	
	const numbers = range(1,5).pipe(skip(3));
	numbers.subscribe(x => console.log(x)); // 4,5

skipLast	//Skip the last count values emitted by the source Observable.
	
	const numbers = range(1,5).pipe(skipLast(3));
	numbers.subscribe(x => console.log(x)); // 1,2

skipUntil	// Returns an Observable that skips items emitted by the source Observable until a second Observable emits an item.
	// In the following example, all emitted values of the interval observable are skipped until
	// the user clicks anywhere within the page.
	
	const intervalObservable = interval(1000);
	const click = fromEvent(document, 'click');
	
	const emitAfterClick = intervalObservable.pipe(
	  skipUntil(click)
	);
	// clicked at 4.6s. output: 5...6...7...8........ or
	// clicked at 7.3s. output: 8...9...10..11.......
	const subscribe = emitAfterClick.subscribe(value => console.log(value));

skipWhile	// Returns an Observable that skips all items emitted by the source Observable as long as a specified condition holds true,
	// but emits all further source items as soon as the condition becomes false.
	
	const numbers = range(1,5).pipe(skipWhile(x=>x<2));
	numbers.subscribe(x => console.log(x)); // 3,4,5

take	// Emits only the first count values emitted by the source Observable.
	// Take the first 5 seconds of an infinite 1-second interval Observable
	
	const intervalCount = interval(1000);
	const takeFive = intervalCount.pipe(take(2));
	takeFive.subscribe(x => console.log(x));
	
	// Logs:
	// 0
	// 1

takeLast	// Emits only the last count values emitted by the source Observable.
	// Take the last 3 values of an Observable from list of values
	
	const many = range(1, 100);
	const lastThree = many.pipe(takeLast(3));
	lastThree.subscribe(x => console.log(x));

takeUntil	// Emits the values emitted by the source Observable until a notifier Observable emits a value.
	// Tick every second until the first click happens
	
	const source = interval(1000);
	const clicks = fromEvent(document, 'click');
	const result = source.pipe(takeUntil(clicks));
	result.subscribe(x => console.log(x));

takeWhile	// Emits values emitted by the source Observable so long as each value satisfies the
	// given predicate, and then completes as soon as this predicate is not satisfied.
	// Emit click events only while the clientX property is greater than 200
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(takeWhile(ev => ev.clientX > 200));
	result.subscribe(x => console.log(x));

throttle	//Emits a value from the source Observable, then ignores subsequent source
	// values for a duration determined by another Observable, then repeats this process.
	// Emit clicks at a rate of at most one click per second
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(throttle(ev => interval(1000)));
	result.subscribe(x => console.log(x));

throttleTime	// Emits a value from the source Observable, then ignores subsequent source values for duration milliseconds, then repeats this process.
	// Emit clicks at a rate of at most one click per second
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(throttleTime(1000));
	result.subscribe(x => console.log(x));

5.Multicasting Operators

multicast

//Returns an Observable that emits the results of invoking a specified selector on items emitted by a ConnectableObservable that shares a single subscription to the underlying stream.

publish

//Returns a ConnectableObservable, which is a variety of Observable that waits until its connect
// method is called before it begins emitting items to those Observers that have subscribed to it.
// Make source$ hot by applying publish operator, then merge each inner observable into a single one and subscribe.

```
onst source$ = zip(interval(2000), of(1, 2, 3, 4, 5, 6, 7, 8, 9)).pipe(
  map(values => values[1])
);

source$
  .pipe(
        publish(multicasted$ =>
          merge(
                multicasted$.pipe(tap(x => console.log('Stream 1:', x))),
                multicasted$.pipe(tap(x => console.log('Stream 2:', x))),
                multicasted$.pipe(tap(x => console.log('Stream 3:', x))),
          )
        )
  )
  .subscribe();

// Results every two seconds
// Stream 1: 1
// Stream 2: 1
// Stream 3: 1
// ...
// Stream 1: 9
// Stream 2: 9
// Stream 3: 9
```

publishLast

// Returns a connectable observable sequence that shares a single subscription to the
// underlying sequence containing only the last notification.
// Similar to publish, but it waits until the source observable completes and stores the last emitted value.
// Similarly to publishReplay and publishBehavior, this keeps storing the last value even if it has no more subscribers.
// If subsequent subscriptions happen, they will immediately get that last stored value and complete.

```
 string connectable =interval(1000)
        .pipe(
          tap(x => console.log("side effect", x)),
          take(3),
          publishLast());

connectable.subscribe(
  x => console.log(  "Sub. A", x),
  err => console.log("Sub. A Error", err),
  () => console.log( "Sub. A Complete"));

connectable.subscribe(
  x => console.log(  "Sub. B", x),
  err => console.log("Sub. B Error", err),
  () => console.log( "Sub. B Complete"));

connectable.connect();

// Results:
//    "side effect 0"
//    "side effect 1"
//    "side effect 2"
//    "Sub. A 2"
//    "Sub. B 2"
//    "Sub. A Complete"
//    "Sub. B Complete"
```

6.Error Handling Operators	

catchError	//Catches errors on the observable to be handled by returning a new observable or throwing an error.
	// Retries the caught source Observable again in case of error, similar to retry() operator
	
	of(1, 2, 3, 4, 5).pipe(
	map(n => {
	if (n === 4) {
	throw 'four!';
	}
	return n;
	}),
	catchError((err, caught) => caught),
	take(30))
	.subscribe(x => console.log(x));
	// 1, 2, 3, 1, 2, 3, ...

retry	//Returns an Observable that mirrors the source Observable with the exception of an error.
	//If the source Observable calls error, this method will resubscribe to the source Observable
	//for a maximum of count resubscriptions (given as a number parameter) rather than propagating the error call.
	
	const source = interval(1000);
	const example = source.pipe(
	  mergeMap(val => {
	if(val > 5){
	  return throwError('Error!');
	}
	return of(val);
	  }),
	  //retry 2 times on error
	  retry(2)
	);
	
	const subscribe = example.subscribe({
	  next: val => console.log(val),
	  error: val => console.log(\`${val}: Retried 2 times then quit!\`)
	});
	
	// Output:
	// 0..1..2..3..4..5..
	// 0..1..2..3..4..5..
	// 0..1..2..3..4..5..
	// "Error!: Retried 2 times then quit!"

retryWhen	

7.Utility Operators	

tap	//Perform a side effect for every emission on the source Observable, but return an Observable that is identical to the source. Map every click to the clientX position of that click, while also logging the click event
	
	const clicks = fromEvent(document, 'click');
	const positions = clicks.pipe(
	  tap(ev => console.log(ev)),
	  map(ev => ev.clientX),
	);
	positions.subscribe(x => console.log(x));

delay	//Delays the emission of items from the source Observable by a given timeout or until a given Date.
	// Delay each click by one second
	
	const clicks = fromEvent(document, 'click');
	const delayedClicks = clicks.pipe(delay(1000)); // each click emitted after 1 second
	delayedClicks.subscribe(x => console.log(x));

delayWhen	// Delays the emission of items from the source Observable by a given time span determined by the emissions of another Observable.
	//Delay each click by a random amount of time, between 0 and 5 seconds
	
	const clicks = fromEvent(document, 'click');
	const delayedClicks = clicks.pipe(

	delayWhen(event => interval(Math.random() * 5000)),
	);
	delayedClicks.subscribe(x => console.log(x));

dematerialize	//Converts an Observable of Notification objects into the emissions that they represent.
	// Convert an Observable of Notifications to an actual Observable
	
	const notifA = new Notification('N', 'A');
	const notifB = new Notification('N', 'B');
	const notifE = new Notification('E', undefined,
	  new TypeError('x.toUpperCase is not a function')
	);
	const materialized = of(notifA, notifB, notifE);
	const upperCase = materialized.pipe(dematerialize());
	upperCase.subscribe(x => console.log(x), e => console.error(e));
	
	// Results in:
	// A
	// B
	// TypeError: x.toUpperCase is not a function

materialize	//Represents all of the notifications from the source Observable as next emissions marked with their original types within Notification objects.
	// Convert a faulty Observable to an Observable of Notifications
	
	const letters = of('a', 'b', 13, 'd');
	const upperCase = letters.pipe(map(x => x.toUpperCase()));
	const materialized = upperCase.pipe(materialize());
	materialized.subscribe(x => console.log(x));
	
	// Results in the following:
	// - Notification {kind: "N", value: "A", error: undefined, hasValue: true}
	// - Notification {kind: "N", value: "B", error: undefined, hasValue: true}
	// - Notification {kind: "E", value: undefined, error: TypeError:
	// x.toUpperCase is not a function at MapSubscriber.letters.map.x
	// [as project] (http://1…, hasValue: false}

observeOn	//Re-emits all notifications from source Observable with specified scheduler.
	// Ensure values in subscribe are called just before browser repaint.
	
	const intervals = interval(10); // Intervals are scheduled
	                                               // with async scheduler by default...
	intervals.pipe(
	  observeOn(animationFrameScheduler), // ...but we will observe on animationFrame
	) // scheduler to ensure smooth animation.
	.subscribe(val => { someDiv.style.height = val + 'px'; });

subscribeOn	//Asynchronously subscribes Observers to this Observable on the specified SchedulerLike.
	const a = of(1, 2, 3, 4).pipe(subscribeOn(asyncScheduler));
	const b = of(5, 6, 7, 8, 9);
	merge(a, b).subscribe(console.log);
	
	/* The output will instead be 5 6 7 8 9 1 2 3 4.
	The reason for this is that Observable b emits its values directly and synchronously
	like before but the emissions from a are scheduled on the event loop because we are now using the async for that specific Observable. */

timeInterval	//Emits an object containing the current value, and the time that has passed between emitting the current value and the previous value, which is calculated by using the provided scheduler's now() method to retrieve the current time at each emission, then calculating the difference. The scheduler defaults to async, so by default, the interval will be in milliseconds.
	//Emit inteval between current value with the last value
	
	const seconds = interval(1000);
	
	seconds.pipe(timeInterval())
	.subscribe(
	        value => console.log(value),
	        err => console.log(err),
	);
	
	seconds.pipe(timeout(900))
	.subscribe(

	value => console.log(value),
	err => console.log(err),
	);
	
	// NOTE: The values will never be this precise,
	// intervals created with `interval` or `setInterval`
	// are non-deterministic.
	
	// {value: 0, interval: 1000}
	// {value: 1, interval: 1000}
	// {value: 2, interval: 1000}

timestamp	// Attaches a timestamp to each item emitted by an observable indicating when it was emitted
	// In this example there is a timestamp attached to the documents click event.
	
	const clickWithTimestamp = fromEvent(document, 'click').pipe(
	timestamp()
	);
	
	// Emits data of type {value: MouseEvent, timestamp: number}
	clickWithTimestamp.subscribe(data => {
	console.log(data);
	});

timeout	// Errors if Observable does not emit a value in given time span.
	// Check if ticks are emitted within certain timespan
	//Use Date to check if Observable completed
	
	let seconds = interval(1000);
	
	seconds.pipe(
	timeout(new Date("December 17, 2020 03:24:00")),
	)
	.subscribe(
	value => console.log(value), // Will emit values as regular `interval` would
	// until December 17, 2020 at 03:24:00.
	err => console.log(err) // On December 17, 2020 at 03:24:00 it will emit an error,
	// since Observable did not complete by then.
	);

timeoutWith	//Errors if Observable does not emit a value in given time span, in case of which subscribes to the second Observable.
	// Add fallback observable
	
	const seconds = interval(1000);
	const minutes = interval(60 * 1000);
	
	seconds.pipe(timeoutWith(900, minutes))
	.subscribe(
	value => console.log(value), // After 900ms, will start emitting `minutes`,
	// since first value of `seconds` will not arrive fast enough.
	err => console.log(err), // Would be called after 900ms in case of `timeout`,
	// but here will never be called.
	);

toArray	//Collects all source emissions and emits them as an array when the source completes.
	
	let source = interval(1000);
	const example = source.pipe(take(10), toArray());
	
	const subscribe = example.subscribe(val => console.log(val));
	// output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

8.Conditional and Boolean Operators	

defaultIfEmpty	//Emits a given value if the source Observable completes without emitting any next value, otherwise mirrors the source Observable.
	// If no clicks happen in 5 seconds, then emit "no clicks"
	
	const clicks = fromEvent(document, 'click');
	const clicksBeforeFive = clicks.pipe(takeUntil(interval(5000)));
	const result = clicksBeforeFive.pipe(defaultIfEmpty('no clicks'));
	result.subscribe(x => console.log(x));

every	//Returns an Observable that emits whether or not every item of the source satisfies the condition specified. A simple example emitting true if all elements are less than 5, false otherwise
	
	of(1, 2, 3, 4, 5, 6).pipe(every(x => x < 5),)
	                              .subscribe(x => console.log(x)); // -> false	

find	//Emits only the first value emitted by the source Observable that meets some condition.
	// Find and emit the first click that happens on a DIV element
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(find(ev => ev.target.tagName === 'DIV'));
	result.subscribe(x => console.log(x));	

findIndex	// Emits only the index of the first value emitted by the source Observable that meets some condition.Emit the index of first click that happens on a DIV element
	
	const clicks = fromEvent(document, 'click');
	const result = clicks.pipe(findIndex(ev => ev.target.tagName === 'DIV'));
	result.subscribe(x => console.log(x));	

isEmpty	//Emits false if the input observable emits any values, or emits true if the input observable completes without emitting any values. Emit false for a non-empty Observable
	
	const source = new Subject<string>();
	const result = source.pipe(isEmpty());
	source.subscribe(x => console.log(x));
	result.subscribe(x => console.log(x));
		source.next('a');
		source.next('b');source.next('c');
		source.complete();
		
		// Results in:
		// a
		// false
		// b
		// c

9.Mathematical and Aggregate Operators	

count	//Counts the number of emissions on the source and emits that number when the source completes.
	// Counts how many odd numbers are there between 1 and 7
	
	const numbers = range(1, 7);
	const result = numbers.pipe(count(i => i % 2 === 1));
	result.subscribe(x => console.log(x));
	// Results :
	// 4

min	interface Person {
	  age: number,
	  name: string
	}
	Observable.of<Person>({age: 7, name: 'Foo'},
	  {age: 5, name: 'Bar'},
	  {age: 9, name: 'Beer'})
	                              .min<Person>((a: Person, b: Person) => a.age < b.age ? -1 : 1)
	                              .subscribe((x: Person) => console.log(x.name)); // -> 'Bar'
	}
	
	or
	Rx.Observable.of(5, 4, 7, 2, 8)
	                          .min()
	                          .subscribe(x => console.log(x)); // -> 2

max	

reduce	// Applies an accumulator function over the source Observable,
	// and returns the accumulated result when the source completes, given an optional seed value.
	//Count the number of click events that happened in 5 seconds
	const clicksInFiveSeconds = fromEvent(document, 'click').pipe(
	  takeUntil(interval(5000)),
	);
	const ones = clicksInFiveSeconds.pipe(mapTo(1));
	const seed = 0;
	const count = ones.pipe(reduce((acc, one) => acc + one, seed));
	count.subscribe(x => console.log(x));