**Angular interview QA**

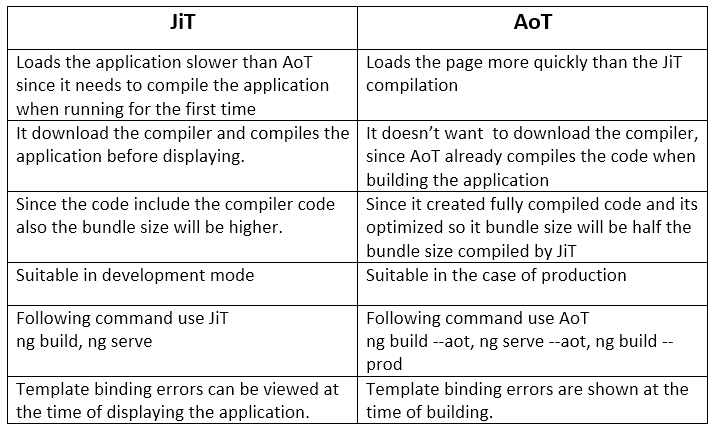
**Q1**. What is the difference between AOT and JIT? What is the advantage of AOT.

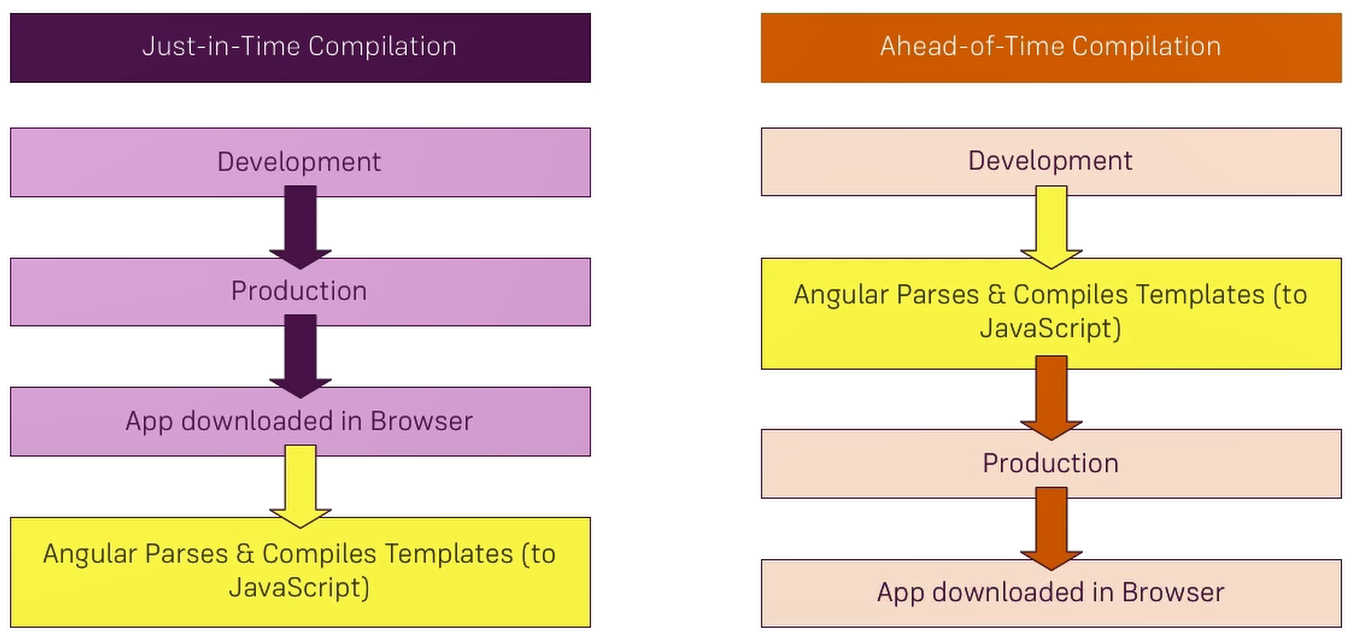
**A**. **JIT** - Compile TypeScript just in time for executing it.

* Compiled in the browser.
* Each file compiled separately.
* No need to build after changing your code and before reloading the browser page.
* Suitable for local development.

**AOT** - Compile TypeScript during build phase.

* Compiled by the machine itself, via the command line (Faster).
* All code compiled together, inlining HTML/CSS in the scripts.
* No need to deploy the compiler (Half of Angular size).
* More secure, original source not disclosed.
* Suitable for production builds.





<https://blog.nrwl.io/angular-is-aot-worth-it-8fa02eaf64d4>

**Q2**. Data sharing among non-related components. If RxJS getter setter then what will happen if refresh the apps.

* This can be done through shared service by using observable's.
* You can use ngrx/store for this. This is similar to Redux arch. You will be getting data from state.

This is a case where you want to use a shared service, as your components are structured as siblings and grandchildren. Start by creating a BehaviorSubject in the service

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

import { BehaviorSubject } from 'rxjs/BehaviorSubject';

@Injectable()

export class DataService {

private messageSource = new BehaviorSubject("default message");

currentMessage = this.messageSource.asObservable();

constructor() { }

changeMessage(message: string) {

this.messageSource.next(message)

}

}

Then inject this service into each component and subscribe to the observable.

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

import { DataService } from "../data.service";

@Component({

selector: 'app-parent',

template: `

{{message}}

`,

styleUrls: ['./sibling.component.css']

})

export class ParentComponent implements OnInit {

message:string;

constructor(private data: DataService) { }

ngOnInit() {

this.data.currentMessage.subscribe(message => this.message = message)

}

}

You can change the value from either component and the value will be updated, even if you don't have the parent/child relationship.

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

import { DataService } from "../data.service";

@Component({

selector: 'app-sibling',

template: `

{{message}}

<button (click)="newMessage()">New Message</button>

`,

styleUrls: ['./sibling.component.css']

})

export class SiblingComponent implements OnInit {

message:string;

constructor(private data: DataService) { }

ngOnInit() {

this.data.currentMessage.subscribe(message => this.message = message)

}

newMessage() {

this.data.changeMessage("Hello from Sibling")

}

}

**Q3.**How to bootstrap angular application.

In Angular there are typically 3 ways of bootstrapping the application :

1. **Default or Automatic Bootstrapping**

**2. Manual Bootstrapping**

**2. Angular Elements ( >=v6 )**

**Default or Automatic Bootstrapping**

It is the default way an angular application bootstraps and main.ts holds the starting point of an application.

platformBrowserDynamic().bootstrapModule(AppModule);

the platformBrowserDynamic() part creates the platform for our app module. An Angular platform is the entry point for Angular on a web page.

the platformBrowserDynamic() part creates the platform for our app module. An Angular platform is the entry point for Angular on a web page.

Angular also has [a ***concept of the running application instance***](https://angular.io/api/core/ApplicationRef) that you can usually inject using ApplicationRef token. There potentially can be many applications on one platform. Each application is created from the module using bootstrapModule method. This is exactly the method that is used in main.ts. So the statement shown in the docs first creates a platform and then the application instance.

Angular can be bootstrapped *in other Javascript host environments asides the browser* (e.g. on the server or in a web worker), thus it is important that we specify the environment in which our App is to be booted.

The bootstrapModule() function helps to bootstrap our root module taking in the root module as its argument. AppModule get processed as per the details provided in the meta-data associated with the AppModule, the dependencies and declarations first processed and resolved before going for the actual rendering on the DOM.

NgModule decorator is the **bootstrap property,**it specifies the component to bootstrap when the application starts. So when all the dependencies and services in the provider is been resolved angular instantiate the bootstrap component and insert into the DOM where you have specified the component selector.

**2.2 Manual Bootstrapping**

Imagine a situation when the component to bootstrap the application is defined by the server during run-time. How can you bootstrap the application later when you get this information about the component ? In such situation manual bootstrapping comes in handy.

First lets create the component :

import { Component } from '@angular/core';  
  
@Component({  
 selector: 'd-comp',  
 template: `<span>I am the dynamic component</span>`  
})  
export class ComponentDynamic {}

Now everything will be same as in the above process except the bootstrapconfiguration**,**you will not specify any component in the bootstrap array just leave it. But we need to specifies the component(s) in the in entryComponentsso that the compiler create factories for them. Angular automatically adds all components specified in the bootstrap property to entry components that is why you usually don’t add the root component to the entryComponents.

@NgModule({  
imports: [ BrowserModule ],  
declarations: [ ComponentDynamic ],  
entryComponents: [ ComponentDynamic ]  
})

Now we don’t specify the component selectors in the index.html, as we want to do it dynamically, so it now it looks like this:

<body>  
 <h1 id="dynamicComponentElement">  
 Loading Dynamic Component content here ...  
 </h1>  
</body>

If you run the application now, you will get the following error:

*The module AppModule was bootstrapped, but it does not declare “@NgModule.bootstrap” components nor a “ngDoBootstrap” method. Please define one of these*

So here Angular complains that we didn’t specify what component should be used for bootstrapping. We will be bootstrapping the app manually later and to do that we need to add ngDoBoostrap method to the AppModule class:

export class AppModule {  
 ngDoBootstrap(app) {   
 // obtain reference to the DOM element that shows status  
 // and change the status to `Loaded`  
 const dynamicComponentElement = document.querySelector('#dynamicComponentElement');  
 dynamicComponentElement.textContent = 'Loaded'; // create DOM element for the component being bootstrapped  
 // and add it to the DOM  
 const componentElement = document.createElement('d-comp');  
 document.body.appendChild(componentElement); // bootstrap the application with the component  
 app.bootstrap(ComponentDynamic);  
 }  
}

in the above snippet first we obtain the DOM element where we want our component to be bootstrapped then we create the element for our dynamic component with the component selector and append it to the body and then finally bootstrap the element appended on the DOM with the respective component using the app.bootstrapmethod by specifying the component as the argument.

**2.3 Angular Elements ( >=v6 )**

The biggest drawback or the headache of the manual bootstrapping method is creating, appending and bootstrapping our element manually and what if we could have a way where we just use the component selector and skip this manual bootstrap part ? If you are having the same thoughts than you are not alone since the release of version 4 this feature is the most anticipated feature in Angular and in version 6 its finally comes as angular-elements.

Angular elements is based on the concept of **web-components**these are a set of features currently being added by the [W3C](https://en.wikipedia.org/wiki/W3C) to the [HTML](https://en.wikipedia.org/wiki/HTML) and [DOM](https://en.wikipedia.org/wiki/Document_Object_Model) specifications that allow for the creation of reusable [widgets](https://en.wikipedia.org/wiki/Web_widget) or components in web documents and web applications.

Lets see how angular elements leverage this

The angular elements feature comes in a different package called @angular/elements which exports a [createCustomElement](https://angular.io/api/elements/createCustomElement)() API that provides a bridge from Angular's component interface and change detection functionality to the built-in DOM API.

Transforming a component to a custom element makes all of the required Angular infrastructure available to the browser. Creating a custom element is simple and straightforward, and automatically connects your component-defined view with change detection and data binding, mapping Angular functionality to the corresponding native HTML equivalents.

Lets go step by step this time :

**Step 1: Add the angular-element and polyfills**

In order to create a custom element you need to add angular-elements to your module and from version 6 of angular CLI you can do it simply using following command :

**ng add** [@angular/elements](http://twitter.com/angular/elements)

This will add the angular element package to you project and also add the polyfills.

**Step 2 : Create a component**

import { Component, Input } from '@angular/core';@Component({  
 selector: 'my-app',  
 templateUrl: './app.component.html',  
 styleUrls: [ './app.component.css' ]  
})export class AppComponent {  
 @Input() name = 'Angular 6';  
}

**Step 3 : Register the component with the module**

@NgModule({  
 imports: [ BrowserModule ],  
 declarations: [ AppComponent ],  
 entryComponents: [ AppComponent ]  
})export class AppModule {  
 constructor(private injector:Injector){} ngDoBootstrap(){  
 **const AppElement = createCustomElement(AppComponent, { injector: this.injector });**  
 **customElements.define('my-app', AppElement);**  
 }  
}

This is the important step: we use the Angular’s createCustomElement function to create a class that can be used with browsers’ native customElements.definefunctionality.

**Step 4: Build**

Now just build it by using the standard ng build command, it will outputs 4 files (runtime.js , scripts.js, polyfills.js and main.js).

**Step 5: Use it**

To use it just include the script(s) to the webpage and insert the custom component selector element to your html. And that’s it.

The build files are bundled with all the require thing your custom component requires including the supporting Angular ecosystem, your module with the custom components and the necessary polyfills. These files contains everything to execute you custom element on any platform. It also takes care of finding the custom element with the selector and bootstrapping of the respective component and rendering it to the DOM.

Every Angular app consists of a file named angular.json. This file will contain all the configurations of the app. While building the app, the builder looks at this file to find the entry point of the application.

"build": {

"builder": "@angular-devkit/build-angular:browser",

"options": {

"outputPath": "dist/angular-starter",

"index": "src/index.html",

"main": "src/main.ts",

"polyfills": "src/polyfills.ts",

"tsConfig": "tsconfig.app.json",

"aot": false,

"assets": [

"src/favicon.ico",

"src/assets"

],

"styles": [

"./node\_modules/@angular/material/prebuilt-themes/deeppurple-amber.css",

"src/style.css"

]

}

}

Inside the build section, the main property of the options object defines the entry point of the application which in this case is main.ts.  
The main.ts file creates a browser environment for the application to run, and, along with this, it also calls a function called **bootstrapModule**, which bootstraps the application. These two steps are performed in the following order inside the **main.ts** file:

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

platformBrowserDynamic().bootstrapModule(AppModule)

In the above line of code, AppModule is getting bootstrapped. The AppModule is declared in the app.module.ts file. This module contains declarations of all the components.

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

entryComponents: [],

bootstrap: [AppComponent]

})

export class AppModule { }

This component is defined in app.component.ts file. This file interacts with the webpage and serves data to it.

The HTML template of the root component is displayed inside the <app-root> tags.

**Q.**What are angular different versions and features.

**Angular 2:**

* 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:**

* Why we don’t have Angular 3?  
  — Angular is being developed in a MonoRepo it means a single repo for everything. [@angular/core](http://twitter.com/angular/core), [@angular/compiler](http://twitter.com/angular/compiler), [@angular/router](http://twitter.com/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.

**Angular 4**

* 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](http://twitter.com/angular/core) to [@angular/animation](http://twitter.com/angular/animation)  
  — don’t import [@animation](http://twitter.com/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:

\*ngIf=”yourCondition; **else myFalsyTemplate**”  
“<ng-template #myFalsyTemplate>Else Html</ng-template>”

**Angular 5**

* 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](http://twitter.com/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://twitter.com/angular/common)/http package.
* Few new Router Life-cycle Events being added

ActivationStart, ActivationEnd, ChildActivationStart, ChildActivationEnd, GuardsCheckStart, GuardsCheckEnd, ResolveStart and ResolveEnd.

* 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

Pending….

**Q.**How to set ngfor and ngif in the same div or tag?

By using ng-container or ng-template

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

<div \*ngFor="let thing of stuff">

{{log(thing)}}

<span>{{thing.name}}</span>

</div>

</ng-container>

*Note: no longer recommended*

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

<div \*ngFor="let thing of stuff">

{{log(thing)}}

<span>{{thing.name}}</span>

</div>

</ng-template>

**Q.**Difference between structural Directive, Attribute directive and components.

**Component**

* It is also a type of directive with **template**,**styles** and **logic** part which is most famous type of directive among all in Angular2. In this type of directive you can use other directives whether it is custom or builtin.

@Component({

selector: "my-app"

directives: [custom\_directive\_here]})

* Use this directive in your view as:

<my-app></my-app>

**Structural directives**

Like **\*ngFor** and **\*ngIf**, used to change the DOM layout by adding and removing DOM elements.

**Attribute directives**

They are used to give custom behavior or style to the existing elements by applying some functions/logic. Like ngStyle is an attribute directive to give style dynamically to the elements.

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

@Directive({

selector: '[Icheck]',

})

export class RadioCheckbox {

// custom logic here...

}

We can use this in the view as shown below:

<span Icheck>HEllo Directive</span>

* If it has a template, it is a Component
* else if it has a selector in brackets "[likethis]", it is an Attribute Directive
* else it is a Structural Directive.

**Q.**Difference between the template and reactive forms.

**Reactive form** can be used in the following situation

* Complex forms with more number of fields.
* Multiple complex validation are there. Custom validations are required
* Require JSON structure to be send with the values in the form.

We can get entire form in a structured way by using **"form.value"**

HTML code will be

<form [formGroup]="form">

First Name <input formControlName="firstName">

Last Name <input formControlName="lastName">

Email <input formControlName="email">

Phone Number <input formControlName="phoneNumber">

</form>

We can get the values from the form like below

{

"firstName": "FName",

"lastName": "LName",

"email": "test@123.com",

"phoneNumber": "123"

}

by calling form.value, where form is FormGroup Variable that we created.

* More flexible, but needs a lot of practice
* Handles any complex scenarios
* No data binding is done (immutable data model preferred by most developers)
* More component code and less HTML markup
* Reactive transformations can be made possible such as
* Handling a event based on a debounce time
* Handling events when the components are distinct until changed
* Adding elements dynamically
* Easier unit testing
* reusable,
* more robust,
* testable,
* more scalable

**Template Driven Form** : It can be used when using simple forms. Like login page. With the two way data binding. We can simply assign value to variable from ui and vice versa.

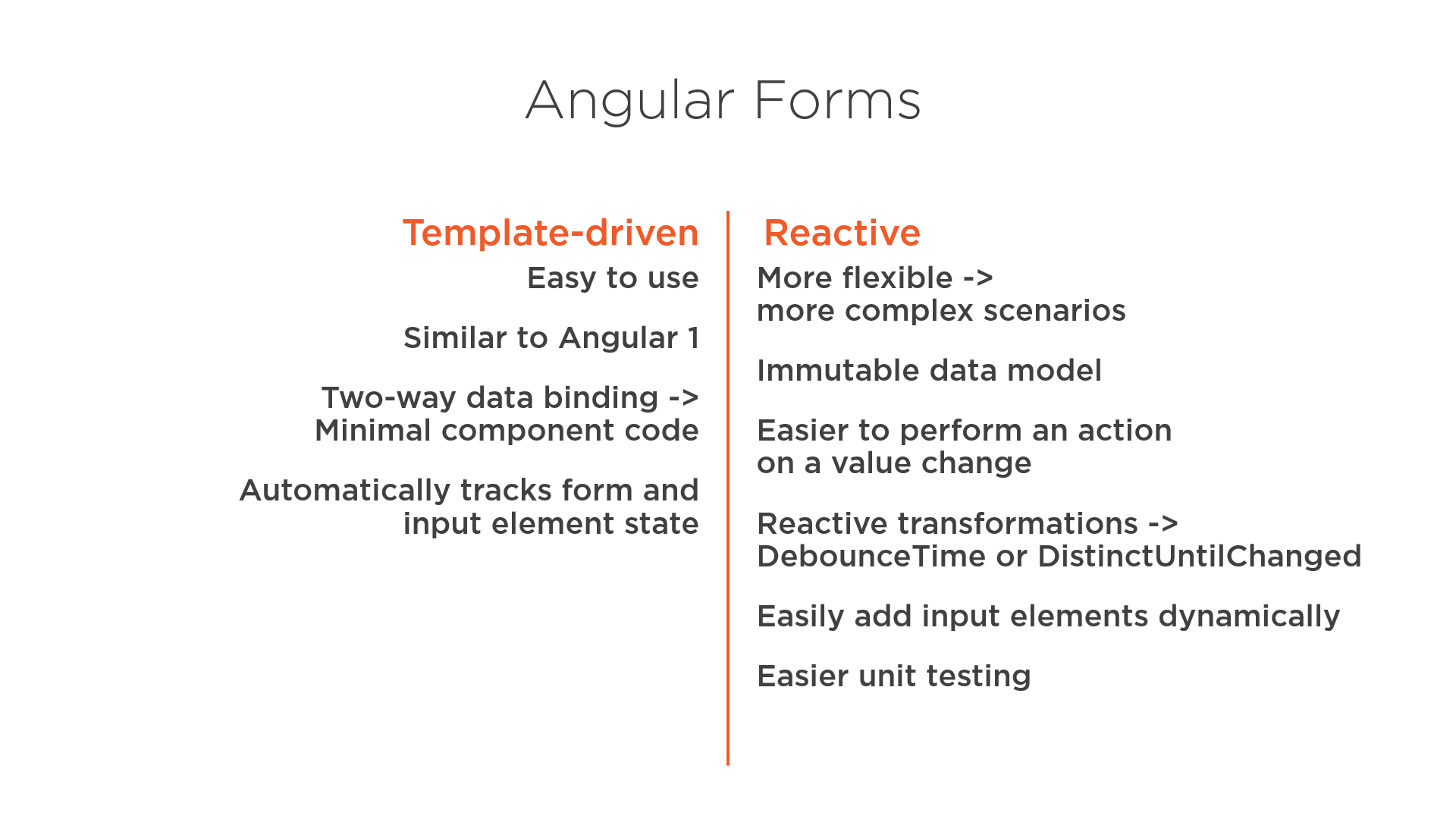
Simple example is if we are giving two way binding for the below input.

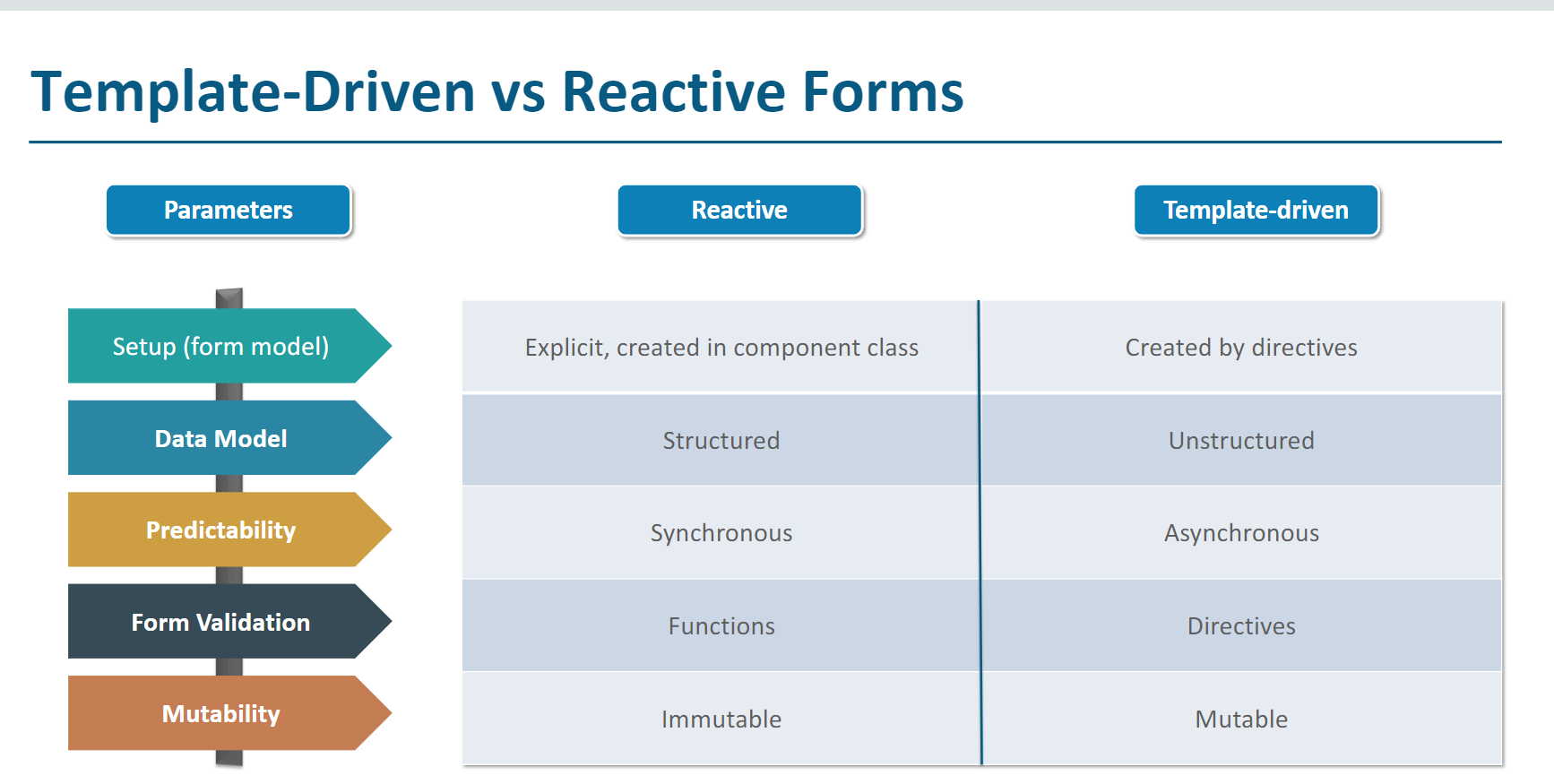
<input [(ngModel)]="username">

We can simply display the value that user is giving in the UI.

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

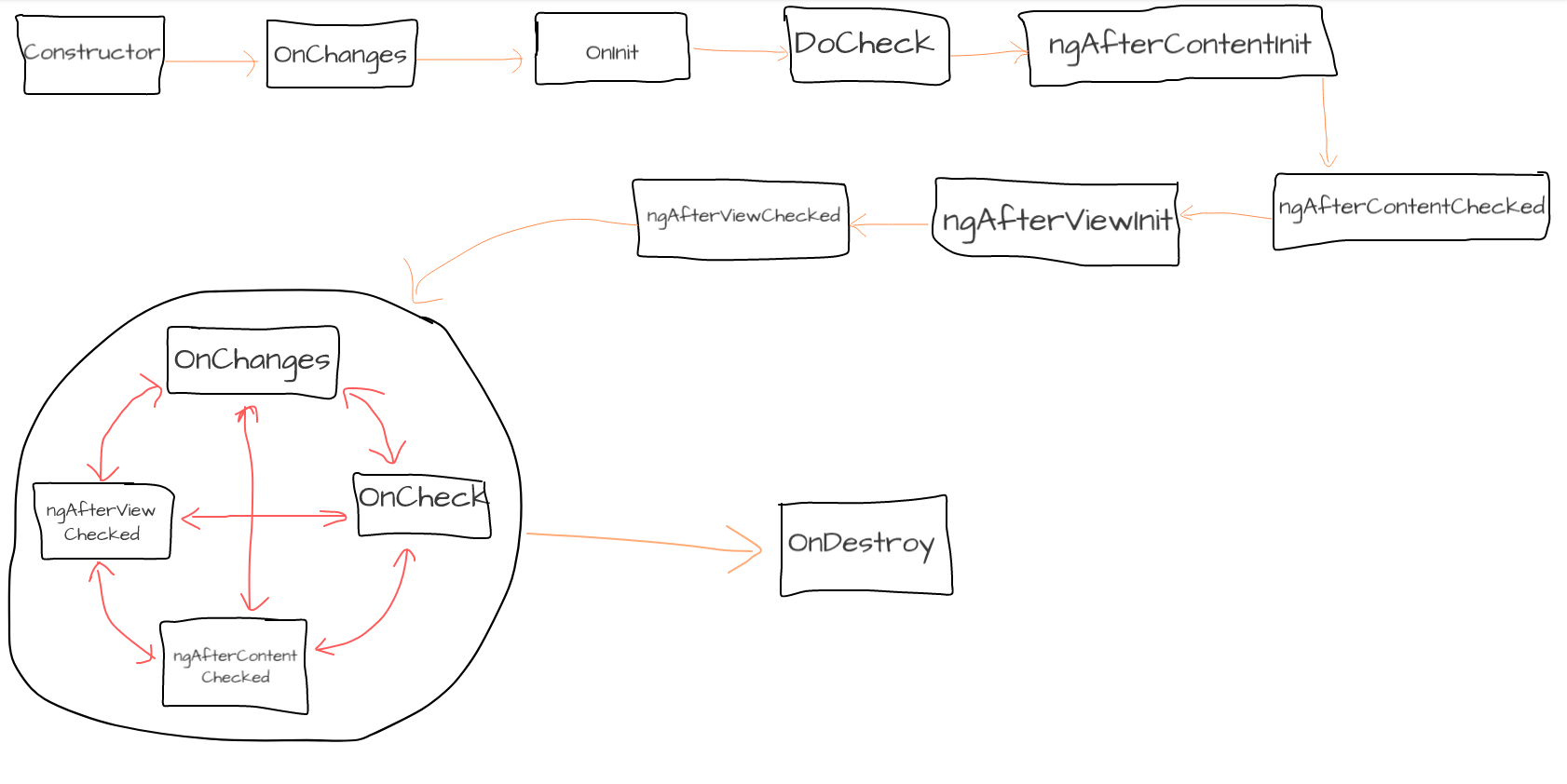
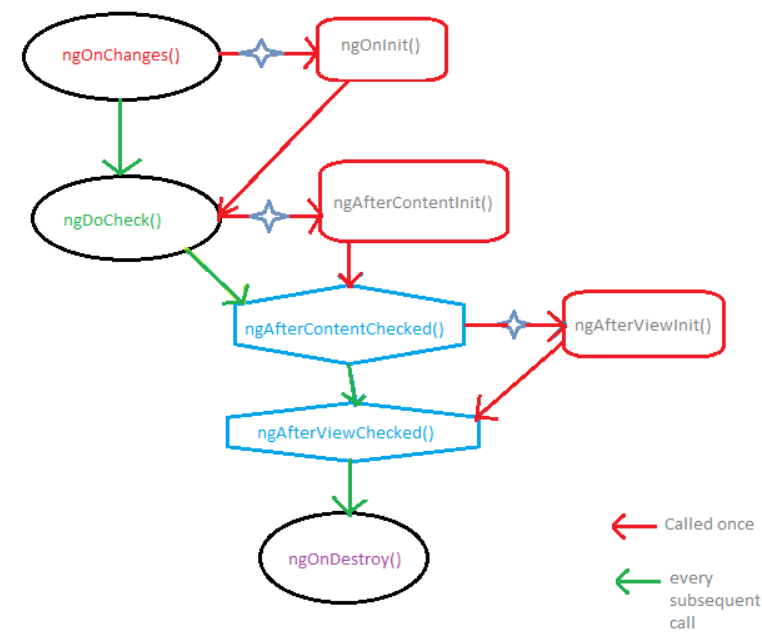
* Easy to use
* Suitable for simple scenarios and fails for complex scenarios
* Two way data binding(using [(NgModel)] syntax)
* Minimal component code
* Automatic track of the form and its data(handled by Angular)
* Unit testing is another challenge





* easy to add,
* less scalable,
* basic form requirements

**Q.** What are all life cycle hook in Angular

* Total 18 life cycle hooks
* 
* 
* 1)**Constructor**: A default method which is called when the class is instantiated.
* 2)**ngOnChanges**: Executes when a new component is created, when one of the bound properties with @Input changes, also it is the only hook that takes an argument when it is called which is called as SimpleChanges.
* 3)**ngOnInit**: Called once the component is initialized. This doesn't allow the component to be visible over the DOM. This runs just after the constructor.
* 4)**ngDoCheck**: Runs when change detection runs. It also runs if there is no change and even if it is just an event that occurred, just in case to make sure if there is something that has changed. (for eg: It will run after a button click event irrespective of that it is making ant change or not)
* 5)**ngAfterContentInit**: This is called after content(ng-content) has been projected into the view.
* 6)**ngAfterContentChecked**: This is called after every projected content has been checked.
* 7)**ngAfterViewInit**: Called after the components view (and child view) has been initialized.
* 8)**ngAfterViewChecked**: Called every time the view (and child view) has been checked.
* 9)**ngOnDestroy**: Called when we generally use an if condition and render the component accordingly. This is mainly called right before the object is destroyed by the angular.

**Q.** Authentication guards, based on token expiration and role-based.

* Angular’s route guards are interfaces which can tell the router whether or not it should allow navigation to a requested route. They make this decision by looking for a true or false return value from a class which implements the given guard interface.
* 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:
* CanActivate
* CanActivateChild
* CanDeactivate
* CanLoad
* Resolve

**Routing Decisions Based on Token Expiration**

// src/app/auth/auth.service.ts

import { Injectable } from '@angular/core';  
import { JwtHelperService } from '@auth0/angular-jwt';

@Injectable()  
export class AuthService {

constructor(public jwtHelper: JwtHelperService) {} // ...  
 public isAuthenticated(): boolean {

const token = localStorage.getItem('token'); // Check whether the token is expired and return  
 // true or false  
 return !this.jwtHelper.isTokenExpired(token);  
 }}

// src/app/auth/auth-guard.service.ts

import { Injectable } from '@angular/core';  
import { Router, CanActivate } from '@angular/router';  
import { AuthService } from './auth.service';

@Injectable()  
export class AuthGuardService implements CanActivate { constructor(public auth: AuthService, public router: Router) {} canActivate(): boolean {  
 if (!this.auth.isAuthenticated()) {  
 this.router.navigate(['login']);  
 return false;  
 }  
 return true;  
 }}

// src/app/app.routes.ts

import { Routes, CanActivate } from '@angular/router';  
import { ProfileComponent } from './profile/profile.component';  
import { AuthGuardService as AuthGuard} from './auth/auth-guard.service';

export const ROUTES: Routes = [  
 { path: '', component: HomeComponent },  
 {   
 path: 'profile',  
 component: ProfileComponent,  
 canActivate: [AuthGuard]   
 },  
 { path: '\*\*', redirectTo: '' }  
];

**Checking for a User’s Role**

// src/app/auth/role-guard.service.ts

import { Injectable } from '@angular/core';  
import {   
 Router,  
 CanActivate,  
 ActivatedRouteSnapshot  
} from '@angular/router';  
import { AuthService } from './auth.service';  
import decode from 'jwt-decode';

@Injectable()  
export class RoleGuardService implements CanActivate { constructor(public auth: AuthService, public router: Router) {} canActivate(route: ActivatedRouteSnapshot): boolean { // this will be passed from the route config  
 // on the data property  
 const expectedRole = route.data.expectedRole; const token = localStorage.getItem('token'); // decode the token to get its payload  
 const tokenPayload = decode(token);

if (  
 !this.auth.isAuthenticated() ||   
 tokenPayload.role !== expectedRole  
 ) {  
 this.router.navigate(['login']);  
 return false;  
 }  
 return true;  
 }}

// src/app/app.routes.ts

import { Routes, CanActivate } from '@angular/router';  
import { ProfileComponent } from './profile/profile.component';  
import {   
 AuthGuardService as AuthGuard   
} from './auth/auth-guard.service';  
import {   
 RoleGuardService as RoleGuard   
} from './auth/role-guard.service';export const ROUTES: Routes = [  
 { path: '', component: HomeComponent },  
 {   
 path: 'profile',   
 component: ProfileComponent,   
 canActivate: [AuthGuard]   
 },  
 {   
 path: 'admin',   
 component: AdminComponent,   
 canActivate: [RoleGuard],   
 data: {   
 expectedRole: 'admin'  
 }   
 },  
 { path: '\*\*', redirectTo: '' }  
];

**Q.** Lazy loading vs eager loading in Angular.

By default angular modules loads eager loading whether they need it or not.

To implement Lazy Loading, following things need to be specified to the RouterModule:

* A route config object with loadChildren instead of component. The value assigned to it would be the relative path to the Lazy Loaded Module, followed by # followed by the name of the module.
* The Lazy loaded module should also implement a routing module in it and in there, it should call **forChild** instead of **forRoot**.

**Q.** What are the common modules and ngmodules automatically added in angular apps? What error it will give if we do not add it.

**Pending…**

<https://medium.com/@cyrilletuzi/understanding-angular-modules-ngmodule-and-their-scopes-81e4ed6f7407>

**Q.** What is a static class and what are the benefits of it.

Static class only creates one instance when module is initialized. Data sharing is very easy along with instances.

**Q.** what is **:host** property in css in Angular application.

if overriding any root component styling then use it. Also, you can use it when using any third party design i.e. Angular Material

Pending….

**Q.** Globalization in Angular and how you handle it.

Pending….

**Q.** View encapsulation in Angular

Pending….

**Q.** How server-side rendering works in Angular

* Pending….

- Facilitate web crawlers through search engine optimization (SEO)

- Improve performance on mobile and low-powered devices

- Show the first page quickly with a first-contentful paint (FCP)

**Q.** How webpack loader works, What is the difference between Webpack and Grunt.

Pending…

**Q.** What is the difference between Subject, BehaviourSubject, and ReplaySubject.

**Observable vs Subject**

The subject is a special type of Observable. Subjects implement observer design pattern which is required to set some value i.e. this.Subject.next(message). Observable always need a subscription and implement the observable pattern. Observable is always unidirectional meaning it flows from source to subscriber. The subject is bidirectional, information flow from the source to the subscriber, and vice versa.

Types of Subject:

**1.BehaviourSubject:** A BehaviourSubject stores the latest value and immediately sends it to all subscribers.

**2. ReplaySubject:**A replay subject stores n number of values and immediately sends it to the subscriber. if we pass 3 in ReplaySubject  then it will cache the last three values on subscription.

private messageSource = new ReplaySubject<string>(3);

**3. AsyncSubject:**Async subject waits until it complete and then sends the final value to the subscriber.

import { AsyncSubject } from ‘rxjs’;

const asyncsub = new AsyncSubject();

asyncsub.subscribe(subs => console.log(subs));

asyncsub.next("Hello 1");

asyncsub.next("Hello Again");

asyncsub.complete();

//Logged only Hello Again //last value will be printed

Async pipes automatically unsubscribe open observables and subject once a component is out of scope.

**Q.** What will happen if error happens in forkJoin.

if you use RxJS6, you need to use **catchError** instead of catch, and **pipe** operators instead of chaining.

// Imports in RxJS6

import {forkJoin, of} from 'rxjs';

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

// Code with pipeable operators in RxJS6

forkJoin(

this.http.post<any[]>(URL, jsonBody1, postJson) .pipe(map((res) => res), catchError(e => of('Oops!'))),

this.http.post<any[]>(URL, jsonBody2, postJson) .pipe(map((res) => res), catchError(e => of('Oops!')))

)

.subscribe(res => this.handleResponse(res))

**Q.** Difference between mergemap, switchMap and concatmap vs flatmap.

**Combining Observables into one Stream:**

**mergeMap:** Creates new observables for any given source. All previous streams/observable keep alive. There is no order in the returned observables, moreover, the order is not preserved. The best use case for mergeMap is when combining a click event with API calls.

**concatMap:** Similar to mergeMap but the order of observables is well preserved. Preserve the order and emits all observable value, works synchronously. Execute slowly because it works synchronously, waits for first observable to complete then only start new observables stream. The best use case is when you are calling an API which gives you id and that id is used in another API.

**switchMap:**Immediately creates new observables and completes the old observables . The best use case for switchMap is search auto-complete. Whenever the user starts typing a new keyword for search, a new observable is created and the old one is completed.

**flatMap:** Immediately creates observables and previous observables are kept alive. flatmap is an alias of mergeMap, mergeMap accepts an optional parameter concurrency, which defines how many Observables can be subscribed at the same time.

**exhaustMap:** Creates observable and waits for it until it complete. All other observable is ignored while waiting for the observable to complete. The best use case is to use the login in the Angular app. Once the user clicks on login then wait until authentication is done! exhaustMap is just the opposite of switchMap. switchMap immediately creates a new observable and completes the old ones however exhaustMap first completes the initial observable and ignores the new ones.

**Q.** What is Promise.all().

This is a key aspect of promises, they're a pipeline, and the data can be massaged by the various handlers in the pipeline.

const promises = [

new Promise(resolve => setTimeout(resolve, 0, 1)),

new Promise(resolve => setTimeout(resolve, 0, 2))

];

Promise.all(promises)

.then(data => {

console.log("First handler", data);

return data.map(entry => entry \* 10);

})

.then(data => {

console.log("Second handler", data);

});

**Q.** Difference between Observable and Subjects.

Pending…

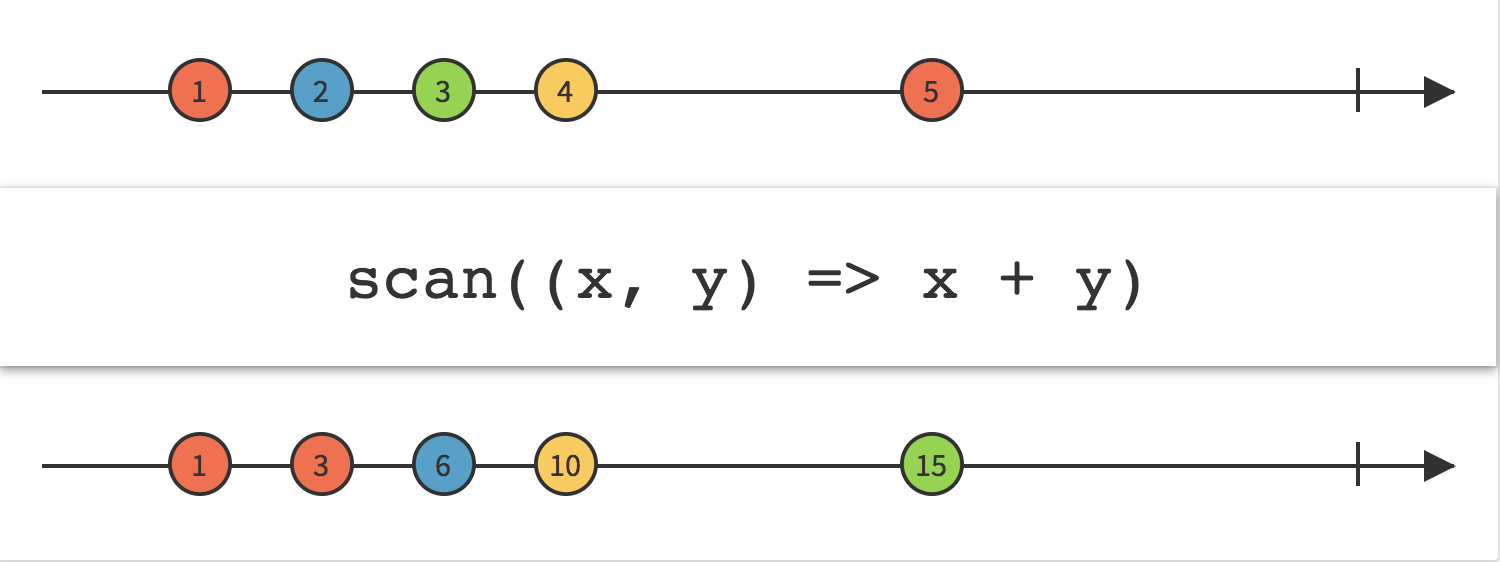
**Q.** Difference between scan and reduce.

Rx.scan is continuous whereas reduce is not continuous.

Scan run sequentially and emits value at each stage, reduce run seq but emit the final value

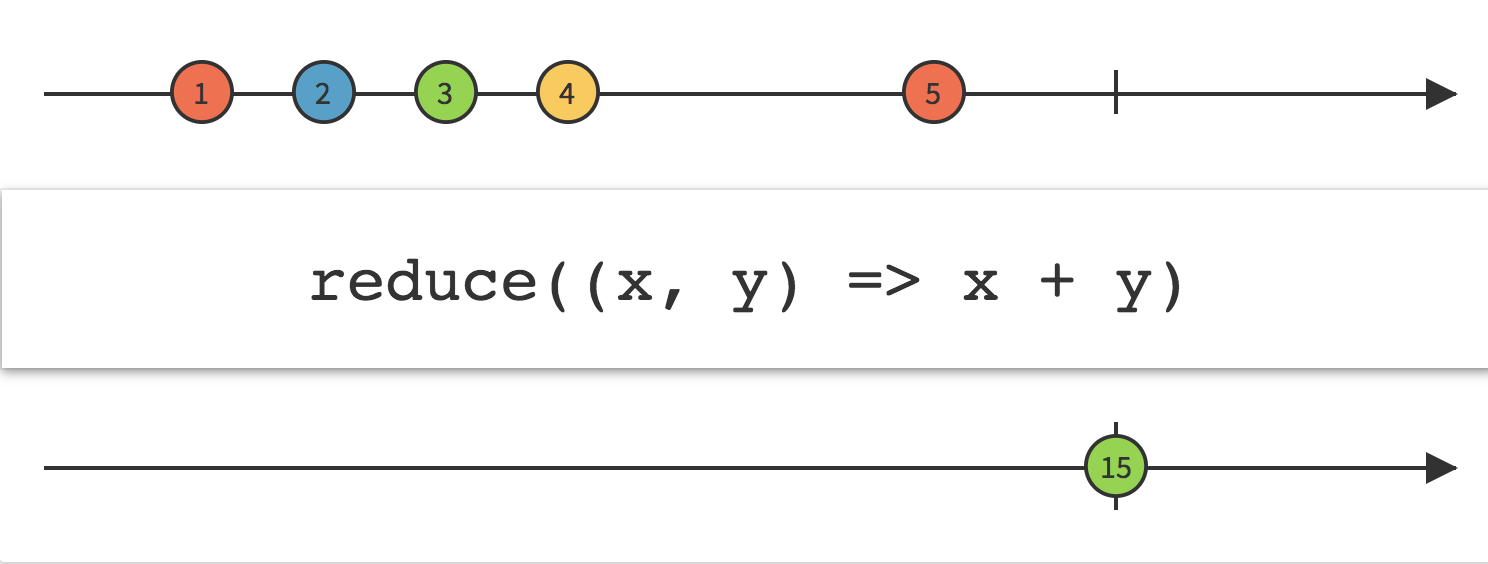
**Scan**

apply a function to each item emitted by an Observable, sequentially, and ***emit each successive value***



**Reduce**

apply a function to each item emitted by an Observable, sequentially, ***and emit the final value***



**Example codes**

**Reduce and Scan**

 import { of } from 'rxjs';

    import { reduce, scan } from 'rxjs/operators';

    of(1, 2, 3, 4).pipe(**reduce**((acc, val) => acc + val)).subscribe(val => console.log('Sum:', val));    //output: Sum: 10'

    of(1, 2, 3, 4).pipe(**scan**((acc, val) => acc + val)).subscribe(val => console.log('Sum:', val));; //output: Sum:1,3,6,10'

**Q.** How to cache data on the client-side.

**Pending…**

**Q.** In which scenario use combineLatest and when to use forkJoin.

**forkJoin:**Calls all observable parallel. Returns all observables as an array once all call is completed. The best use case is when you want to call APIs that are not depends on each other.

**Ex.** Multiple uploads

we can combine insert and update APIs in forkJoin based on conditions.

const promiseList = [];

promiseList.push(this.myService.getData1());

promiseList.push(this.myService.getData2());

forkJoin(promiseList).subscribe(resultList=>{

//Initialize component data here

this.dataModel.data1= resultList[0]

})

**combineLatest:** Begins when all observables fired at least once. Afterward it fires when any of the event changes.

The best use case in an Angular application is combining a data table and dropdown events action.

products$ =combineLatest([this.productService.product$,this.action$])

.pipe(map(([products,category])=>

products.filter(x=>x.categoryId===category)

));

Once merged with combined with the latest whenever the user selects a product category dropdown data table data automatically filtered. No need to write any event in the Angular component. This is sometimes called a reactive style of development.

Now we can use mergeMap while filtering the selectedCategory. If the user clicks on the drop-down again and the previous request is not completed, then switchMap will cancel the previous request and start a new Observables.

selectedCategoryCliks$ = this.selectedCategory$

.pipe(

filter(selectedCategory => Boolean(selectedCategory)),

switchMap(selectedCategory =>

from(selectedCategory.Id)

.pipe(

mergeMap(Id => this.http.get<Interface>(`${environment.getURL}`)),

toArray(),

tap(x => console.log(‘’)

)

)

);

**Q.** Scope in javascript

Scoping is determining where variables, functions, and objects are accessible in your code during runtime. This means the scope of a variable(where it can be accessed) is controlled by the location of the variable declaration.

In Javascript, there are two scopes:

**Global Scope**

There is only one Global scope throughout a JavaScript document. A variable is in the Global scope if it’s defined outside of a function.

**Local Scope**

Variables declared within a function are in the local scope. Local scope is also called function scope because local scope is created by functions in Javascript. Variables in the local scope are only accessible within the function in which they are defined,

***Variable Shadowing***

In JavaScript, variables with the same name can be specified at multiple layers of nested scope. In such case *local variables gain priority* over global variables. This is called ***shadowing****.*

**Q.** Closures in Javascript.

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

**Q.** Difference between const and let and diff between var vs let.

Const value will not change in the scope provided. But array or object property can be added later also.  
Let value depends on the scope.  
Var value will be same in function.

**Scoping rules**

Main difference is scoping rules. Variables declared by var keyword are scoped to the immediate function body (hence the function scope) while let variables are scoped to the immediate *enclosing* block denoted by { } (hence the block scope).

-The reason why let keyword was introduced to the language was function scope is confusing and was one of the main sources of bugs in JavaScript

**Hoisting**

**Hoisting** in JavaScript is a feature in which the interpreter moves the function and variable declarations to the top of their containing [scope](https://medium.com/@lenafaure/understand-javascript-variable-scope-with-ease-221a6d41dc43).

Note that hoisting only moves the declaration and not the assignment.

* Functions are hoisted first, and then variables.
* Function declarations have priority over variable declarations, but not over variable assignments.

While variables declared with **var** keyword are [hoisted](https://dev.to/godcrampy/the-secret-of-hoisting-in-javascript-egi) (*initialized with****undefined****before the code is run*) which means they are accessible in their enclosing scope even before they are declared:

function run() {

console.log(foo); // **undefined**

var foo = "Foo";

console.log(foo); // Foo

}

run();

**let** variables are not initialized until their definition is evaluated. Accessing them before the initialization results in a ***ReferenceError***.

function checkHoisting() {

console.log(foo); // ReferenceError

let foo = "Foo";

console.log(foo); // Foo

}

checkHoisting();

**Q.** What is the use of strict in javascript, Why it is used and what is the benefits of using strict give one example.

* Duplicate keys in object.
* Variables without var
* Duplicate arguments
* Freezes the arguments of the functions

allows you to place a program, or a function, in a “strict” operating context. This strict context prevents certain actions from being taken and throws more exceptions. The statement “use strict”; instructs the browser to use the Strict mode.

**Benifits of using use strict:**

Strict mode eliminates some JavaScript silent errors by changing them to throw errors.

Strict mode fixes mistakes that make it difficult for JavaScript engines

It prevents, or throws errors, when relatively “unsafe” actions are taken (such as gaining access to the global object).

It disables features that are confusing or poorly thought out.

Strict mode makes it easier to write “secure” JavaScript.

**How to use strict mode:** Strict mode can be used in two ways, remember strict mode doesn’t work with block statements enclosed in {} braces.

Used in global scope for the entire script.

It can be applied to individual functions.

**Using Strict mode for the entire script:** To invoke strict mode for an entire script, put the exact statement “use strict”; (or ‘use strict’;) before any other statements.

// Whole-script strict mode syntax

'use strict';

let v = "strict mode script!";

**Using Strict mode for a function:** Likewise, to invoke strict mode for a function, put the exact statement “use strict”; (or ‘use strict’;) in the function’s body before any other statements.

function strict() {

// Function-level strict mode syntax

'use strict';

function nested() { return 'Javascript on GeeksforGeeks'; }

return "strict mode function! " + nested();

}

function notStrict() { return "non strict function"; }

**Examples of using Strict mode:**

**Example:** In normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

// Using a variable, without declaring it, is not allowed:

'use strict';

 x = 3.14;  // will throw an error

Example:**Using strict mode, don’t allow to use a variable without declaring it.**

// Objects are variables too.

// Using an object, without declaring it, is not allowed:

'use strict';

  // Will throw an error

  x = {p1:10, p2:20};

**Example:** Deleting a variable (or object) and a function is not allowed.

'use strict';

 let x = 3.14;

// Deleting a function is also not allowed

'use strict';

 function x(p1, p2) {};

 // Will throw an error

 delete x;

Example:**Duplicating a parameter name is not allowed.**

'use strict';

 // Will throw an error

 function x(p1, p1) {};

**Example:** Octal numeric literals are not allowed.

'use strict';

 // Will throw an error

 let x = 010;

**Example:** Escape characters are not allowed.

'use strict';

 // Will throw an error

 let x = \010;

Example:**Writing to a read-only property is not allowed.**

'use strict';

 let obj = {};

 Object.defineProperty(obj, "x", {value:0, writable:false});

 // Will throw an error

 obj.x = 3.14;

**Example:** Writing to a get-only property is not allowed.

'use strict';

 let obj = {get x() {return 0} };

 // Will throw an error

 obj.x = 3.14;

**Example:** Deleting an undeletable property is not allowed.

'use strict';

 // Will throw an error

 delete Object.prototype;

**Example:** The string “eval” cannot be used as a variable.

'use strict';

 // Will throw an error

 let eval = 3.14;

**Example:** The string “arguments” cannot be used as a variable.

'use strict';

 // Will throw an error

 let arguments = 3.14;

**Example:** The with statement is not allowed.

'use strict';

 // Will throw an error

 with (Math){x = cos(2)};

**Q.** Difference between ES5 and ES6?

Callback vs Promise  
Arrow functions  
Exporting and importing modules  
Type in which we assign objects

**Reducing code with Arrow Function**

Arrow functions brought a lot of clarity & code reduction to Javascript. Let’s take a look at different ways we can define function now.

Here is the ES5 version

function greetings (name) {  
 return 'hello ' + name  
}

Now take a look at different ways we can define function in ES6

const greetings = (name) => {  
 return `hello ${name}`;  
}

const greetings = name => `hello ${name}`;

Another thing to notice that we don’t have to write the return keyword to return the computed value

computed expression automatically get returned when the function will be executed.

**Manipulating objects in ES6 vs ES5**

object destructuring and rest/spread operators made it working with objects very easy now.

merge two objects in ES5

var obj1 = { a: 1, b: 2 }  
var obj2 = { a: 2, c: 3, d: 4}  
var obj3 = Object.assign(obj1, obj2)

We have to merge the object using Object.assign() which takes both objects as input and outputs the merged object.

in ES6

const obj1 = { a: 1, b: 2 }  
const obj2 = { a: 2, c: 3, d: 4}  
**const obj3 = {...obj1, ...obj2}**

object destructuring

If you have to extract multiple values from ES5 you have to write 3–4 lines of code like this:

var obj1 = { a: 1, b: 2, c: 3, d: 4 }  
var a = obj1.a  
var b = obj1.b  
var c = obj1.c  
var d = obj1.d

ES6 here to rescue us

const obj1 = { a: 1, b: 2, c: 3, d: 4 }  
const {a,b,c,d} = obj1

We define a object like this in ES5

var a = 1  
var b = 2  
var c = 3  
var d = 4var obj1 = { a: a, b: b, c: c, d: d }

In ES6 you will do something like this:

var a = 1  
var b = 2  
var c = 3  
var d = 4var obj1 = { a, b, c, d }

**Promises vs Callbacks**

Javascript is an Async language we all know that. This feature gives us a lot of freedom when we write code. We have a non-blocking architecture in our hand because of which we can write non-dependent code easily.

function isGreater (a, b, cb) {   
 var greater = false  
 if(a > b) {  
 greater = true  
 }  
 cb(greater)

}

isGreater(1, 2, function (result) {  
 if(result) {  
 console.log('greater');  
 } else {  
 console.log('smaller')  
 }  
})

how ES6 have handled this.

const isGreater = (a, b) => {  
 return new Promise ((resolve, reject) => {  
 if(a > b) {  
 resolve(true)  
 } else {  
 reject(false)  
 }  
 })  
}

isGreater(1, 2)  
 .then(result => {  
 console.log('greater')  
 })  
 .catch(result => {  
 console.log('smaller')  
 })

**Exporting & Importing Modules**

exporting and importing module syntax changed completely with the introduction of ES6 specification.

ES5 version

var myModule = require('./myModule');

ES6 version

import myModule from './myModule';

When you export something using default we will import a module like this.

import myModule from './myModule';

But ES6 also provides us with an ability to export and import multiple child modules or variables from a single module.

So in your module file you will export your module something like this

export const x = 1;  
export const y = 2;  
export const z = 'String';

And import them something like this

import {x, y, z} from './myModule';

**Q.** How many ways you can create objects in javascript?

JavaScript is an object-based language based on prototypes, rather than being class-based.

**Creating object with a constructor:**

One of the easiest way to instantiate an object in JavaScript. Constructor is nothing but a function and with help of new keyword, constructor function allows to create multiple objects of same flavor

function vehicle(name,maker,engine){

    this.name = name;

    this.maker = maker;

    this.engine = engine;

}

//new keyword to create an object

let car  = new vehicle('GT','BMW','1998cc');

//property accessors

console.log(car.name);

console.log(car.maker);

console.log(car['engine']);

**Using object literals:**

Literals are smaller and simpler ways to define objects.We simple define the property and values inside curly braces as shown below:

//creating js objects with object literal

let car = {

    name : 'GT',

    maker : 'BMW',

    engine : '1998cc'

};

//property accessor

console.log(car.name); //dot notation

console.log(car['maker']); //bracket notation

**Creating object with Object.create() method:**

The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.

const coder = {

    isStudying : false,

    printIntroduction : function(){

        console.log(`My name is ${this.name}. Am I studying?: ${this.isStudying}`);

    }

};

const me = Object.create(coder);

me.name = 'Mukul';

me.isStudying = true;

me.printIntroduction();

**Using es6 classes:**

ES6 supports class concept like any other Statically typed or object oriented language. So, object can be created out of a class in javascript as well as shown below:

//using es6 classes

class Vehicle {

  constructor(name, maker, engine) {

    this.name = name;

    this.maker =  maker;

    this.engine = engine;

  }

}

let car1 = new Vehicle('GT', 'BMW', '1998cc');

console.log(car1.name);  //GT

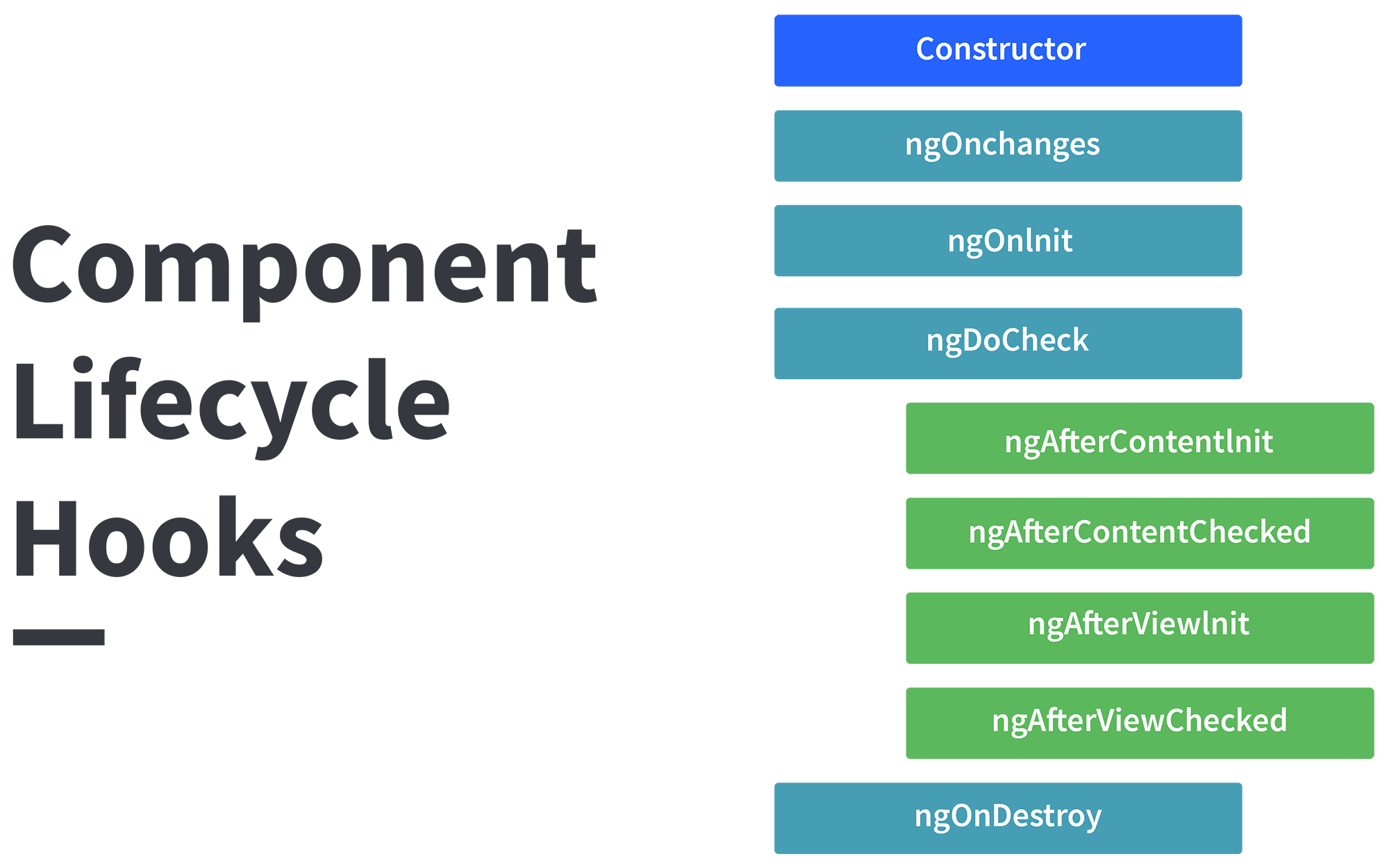
Q. What are some of the advantages of Angular over other frameworks?

Features that are provided out of the box - Angular provides a number of built-in features like,**routing**, **state management**, **rxjs library** **and http services** straight out of the box. This means that one does not need to look for the above stated features separately. They are all provided with angular.

* Declarative UI - Angular uses HTML to render the UI of an application. HTML is a declarative language and is much easier to use than JavaScript.
* Long-term Google support - Google announced Long-term support for Angular. This means that Google plans to stick with Angular and further scale up its ecosystem.

Q. What are lifecycle hooks in Angular? Explain a few lifecycle hooks.

Every component in Angular has a lifecycle, different phases it goes through from the time of creation to the time it's destroyed. Angular provides **hooks** to tap into these phases and trigger changes at specific phases in a lifecycle.



**ngOnChanges( )** This hook/method is called before ngOnInit and whenever one or more input properties of the component changes.  
This method/hook receives a SimpleChanges object which contains the previous and current values of the property.

**ngOnInit( )** This hook gets called once, after the ngOnChanges hook.  
It initializes the component and sets the input properties of the component.

**ngDoCheck( )** It gets called after ngOnChanges and ngOnInit and is used to detect and act on changes that cannot be detected by Angular.  
We can implement our change detection algorithm in this hook.

 ngAfterContentInit( ) It gets called after the first ngDoCheck hook. This hook responds after the content gets projected inside the component.

**ngAfterContentChecked( )** It gets called after ngAfterContentInit and every subsequent ngDoCheck. It responds after the projected content is checked.

**ngAfterViewInit( )** It responds after a component's view, or a child component's view is initialized.

**ngAfterViewChecked( )** It gets called after ngAfterViewInit, and it responds after the component's view, or the child component's view is checked.

**ngOnDestroy( )** It gets called just before Angular destroys the component. This hook can be used to clean up the code and detach event handlers.

Q.Explain string interpolation and property binding in Angular.

String interpolation and property binding are parts of data-binding in Angular. Data-binding is a feature in angular, which provides a way to communicate between the component(Model) and its view(HTML template).  
Data-binding can be done in two ways, **one-way** binding and **two-way** binding.

String interpolation and property binding allow only one-way data binding.

**String interpolation** uses the double curly braces **{{ }}** to display data from the component.

Using property binding, we can bind the DOM properties of an HTML element to a component's property. **Property binding** uses the square brackets **[ ]** syntax.