Angular 2/4 Question -

**1 - diff between promise and observable with syntax**

[**http://csharp-video-tutorials.blogspot.in/2017/09/angular-promises-vs-observables.html**](http://csharp-video-tutorials.blogspot.in/2017/09/angular-promises-vs-observables.html)

In [Angular 2](https://www.youtube.com/watch?v=WWQZCDegWHg&list=PL6n9fhu94yhWqGD8BuKuX-VTKqlNBj-m6), to work with asynchronous data we can use either Promises or Observables. In our previous videos in this series, we discussed using both Observables and Promises. There are several differences between Promises and Observables. In this video let's discuss these differences.  
  
  
As a quick summary the differences are shown in the table below

|  |  |
| --- | --- |
| **Promise** | **Observable** |
| Emits a single value | Emits multiple values over a period of time |
| Not Lazy | Lazy. An Observable is not called until we subscribe to the Observable |
| Cannot be cancelled | Can be cancelled using the unsubscribe() method |
|  | Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc. |

**A Promise emits a single value where as an Observable emits multiple values over a period of time**. You can think of an Observable like a stream which emits multiple items over a period of time and the same callback function is called for each item emitted. So with an Observable we can use the same API to handle asynchronous data whether that data is emitted as a single value or multiple values over a period of time.   
  
**A Promise is not lazy where as an Observable is Lazy**. Let's prove this with an example. Consider this method getEmployeeByCode() in employee.service.ts. Notice this method returns an Observable. 

getEmployeeByCode(empCode: string): Observable<IEmployee> {

    return this.\_http.get("http://localhost:31324/api/employees/" + empCode)

        .map((response: Response) => <IEmployee>response.json())

        .catch(this.handleError);

}

Here is the consumer code of the above service. In our example, this code is in employee.component.ts in ngOnInit() method. Notice we are subscribing to the Observable using the subscribe() method. An Observable is lazy because, it is not called and hence will not return any data until we subscribe using the subscribe() method. At the moment we are using the subscribe() method. So the service method should be called and we should receive data.

ngOnInit() {

    let empCode: string = this.\_activatedRoute.snapshot.params['code'];

    this.\_employeeService.getEmployeeByCode(empCode)

        .subscribe((employeeData) => {

            if (employeeData == null) {

                this.statusMessage =

                    'Employee with the specified Employee Code does not exist';

            }

            else {

                this.employee = employeeData;

            }

        },

        (error) => {

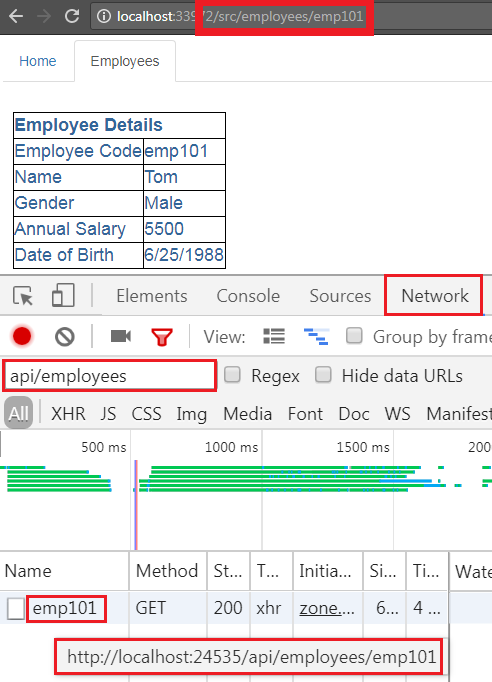
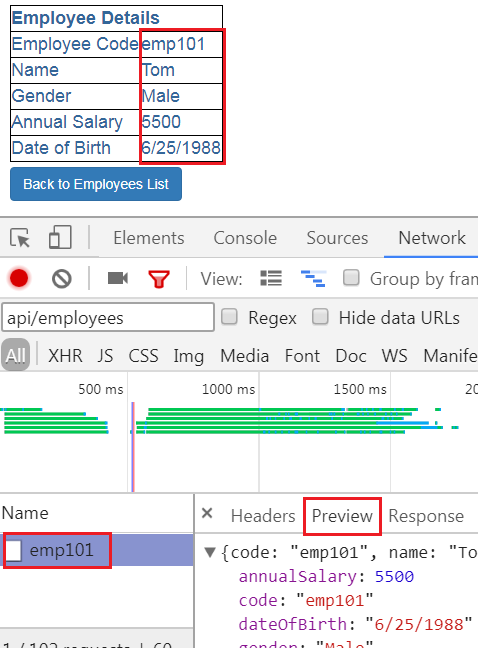
            this.statusMessage =

                'Problem with the service. Please try again after sometime';

            console.error(error);

        });

}

**To prove this**  
1. Launch Browser developer tools by pressing F12 while you are on the browser.   
2. Navigate to /src/employees/emp101  
3. Click on the **Network Tab**  
4. In the Filter textbox, type "api/employees"  
5. At this point, you should only see the request issued to EmployeeService in the table below  
6 Hover the mouse over "emp101" under "Name" column in the table  
7. Notice the request to employee service (/api/employees/emp101) is issued   
  
   
  
Instead of hovering over the request, if you click on it, you can see the response data as shown below. Make sure you select the "preview" tab.   
  
   
  
Now in **employee.component.ts** file, comment the subscribe() method code block as shown below. Notice we are still calling the getEmployeeByCode() method of the EmployeeService. Since we have not subscribed to the Observable, a call to the EmployeeService will not be issued over the network.

ngOnInit() {

    let empCode: string = this.\_activatedRoute.snapshot.params['code'];

    this.\_employeeService.getEmployeeByCode(empCode)

        //.subscribe((employeeData) => {

        //    if (employeeData == null) {

        //        this.statusMessage =

        //            'Employee with the specified Employee Code does not exist';

        //    }

        //    else {

        //        this.employee = employeeData;

        //    }

        //},

        //(error) => {

        //    this.statusMessage =

        //        'Problem with the service. Please try again after sometime';

        //    console.error(error);

        //});

}

With the above change in place, reload the web page. Notice no request is issued over the network to the EmployeeService. You can confirm this by looking at the network tab in the browser developer tools. So this proves that an Observable is lazy and won't be called unless we subscribe using the subscribe() method.  
  
Now, let's prove that a Promise is NOT Lazy. I have modified the code in employee.service.ts to return a Promise instead of an Observable. The modified code is shown below.

getEmployeeByCode(empCode: string): Promise<IEmployee> {

    return this.\_http.get("http://localhost:31324/api/employees/" + empCode)

        .map((response: Response) => <IEmployee>response.json())

        .toPromise()

        .catch(this.handlePromiseError);

}

Here is the consumer code of the above service. In our example, this code is in employee.component.ts in ngOnInit() method. 

ngOnInit() {

    let empCode: string = this.\_activatedRoute.snapshot.params['code'];

    this.\_employeeService.getEmployeeByCode(empCode)

        .then((employeeData) => {

            if (employeeData == null) {

                this.statusMessage =

                    'Employee with the specified Employee Code does not exist';

            }

            else {

                this.employee = employeeData;

            }

        },

        (error) => {

            this.statusMessage =

                'Problem with the service. Please try again after sometime';

            console.error(error);

        });

}

Because a promise is eager(not lazy), calling employeeService.getEmployeeByCode(empCode) will immediately fire off a request across the network to the EmployeeService. We can confirm this by looking at the network tab in the browser tools.   
  
Now you may be thinking, then() method in this case is similar to subscribe() method. If we comment then() method code block, will the service still be called. The answer is YES. Since a Promise is NOT LAZY, Irrespective of whether you have then() method or not, calling employeeService.getEmployeeByCode(empCode) will immediately fire off a request across the network to the EmployeeService. You can prove this by commenting then() method code block and reissuing the request.   
  
In our next video, we will discuss how to retry when a request fails and in the video after that we will discuss how cancel an Observable. 

**2 - what is component**

https://medium.com/front-end-hacking/angular-2-component-lifecycle-hooks-fa5a84b4b64d

May 5, 2016

# Angular 2 — Component Lifecycle Hooks

Each Angular application goes through a lifecycle. In fact, Angular 2 is built on three components, so each component goes through its own lifecycle as well.

This is quite advanced stuff and I go through this it a high level here, so any questions please ask!

As you already know, all the components have their own constructor; it is important however, to remember that the way to access the inputs of a component is not via its constructor.

If you want to access the value of an input - to load additional data from the server for example - you have to use a lifecycle phase.

*The phases available are:*

### **Constructor:**

The constructor of the component class is called before any other component lifecycle hook. If our component is based on any dependencies, the constructor is the best place to inject those dependencies.

Example:

import {Component} from 'angular2/core';  
import {CarService} from './carService';

@Component({  
 selector: ‘list-cars’,  
 template: `  
 <ul>  
 <li \*ngFor="#car of cars">{{car.name}}</li>  
 </ul>  
 `  
})  
class AppComponent {  
 cars:Array<any>;  
 constructor(**private \_carService:** CarService) {  
 this.cars = \_carService.getCars();  
 }  
}

### ngOnInit:

The ngOnInit method of a component is called directly **after**the constructor and **after**the ngOnChange is triggered for the first time. It is the perfect place for initialisation work.

### ngOnChanges:

The ngOnChanges will be called first when the value of a bound property changes. It executes, every time the value of an input property changes. It will receive a changes map, containing the current and previous values of the binding, wrapped in a SimpleChange.

{"brand":{"**previousValue**":"","**currentValue**":"BMW"}}

In the case above, one change to the input property brand is reported. The value of this property has been changed from an empty string to the string “BMW”.

### ngOnDestroy:

The ngDestroy is called in a component’s lifecycle just before the instance of the component is finally destroyed. It is the perfect place to clean the component — for example, to cancel background tasks.

**Quick example:**

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

export class OnDestroyDirective implements OnDestroy {  
 sayHello: number;  
 constructor() {  
 this.sayHiya = window.setInterval(() => console.log('hello'), 1000);  
 }  
 **ngOnDestroy**() {  
 window.clearInterval(this.sayHiya);  
 }  
}

If we do not use the ngOnDestroy method we will have the thread logging “hello” until the end or it crashes….

*More advanced phases are:*

### ngDoCheck:

ngDoCheck is triggered every time the input properties of a component or a directive are checked. We can use this lifecycle hook to extend the check with our own custom check logic. It can also be useful if we want to accelerate the change detection by checking the bare minimum and not using the default algorithm (although we usually do not use this).

### ngAfterContentInit:

The ngAfterContentInit lifecycle hook is called **after ngOnInit**when the component or directive’s content has been initialised; basically when all the bindings of the component have been checked for the first time.

### ngAfterContentChecked:

Called **after every**check of the component or directive’s content, effectively when all the bindings of the components have been checked; even if they haven’t changed.

### ngAfterViewInit:

Called **after** **ngAfterContentInit**when the component’s view has been initialised. Applies to components only.

### ngAfterViewChecked:

Called **after every**check of the component’s view. Applies to components only. When all the bindings of the children directives have been checked; even if they haven’t changed. It can be useful if the component is waiting for something coming from its child components.

**3 - Diff angular1 and 2**

Called as AngularJS 1.x and Angular. Components are used instead of Controllers and $scope. A component is a class with its own data and methods. Option to write code in different languages. Designed for Speed. Supposed to be 5 times faster than Angular 1. Designed for Mobile development also. More modular. It is broken into many packages. Data binding is done with no new directives. We bind to attributes of html elements. Event handling is done with DOM events and not directives. Simpler API   
  
<http://csharp-video-tutorials.blogspot.in/2017/09/difference-between-angularjs-angular-2.html>

**4 - what is component**

**A:**In angular, a Component is a special kind of directive that uses a simpler configuration which is suitable for a component-based application structure.  
  
**in details**  
Make a folder called **app** inside our project folder.  
Then create a **app.component.ts** file inside that app folder and copy/paste with the following snippet:

// Angular2-helloworld-example/app/app.component.ts

import { Component } from ' Angular2/core';

@Component({

selector: 'my-app',

templateUrl: 'app/app.html'

})

export class AppComponent {

target: string;

constructor() {

this.target = 'First angular App';

}

}

#### **Whatâ€™s a component?**

In Angular 2, components are a fundamental concept. It is the way we define views and control the logic on the page. We passed in a configuration object to the component decorator. This object has two properties : selector and template. The selector is the HTML element that angular will looking for. Every times it founds one, angular will instantiate a new instance of our AppComponent class, and place our template.

As you may also notice we export our class at the end. All classes in TypeScript have a constructor, whether you specify one or not. If you do not define the constructor, the compiler will automatically add one. The constructor is called before any other component lifecyle hook. If the component has any dependencies, the constructor is the best place to inject those dependencies. The export statement tells TypeScript that this is a module whose AppComponent class is public and accessible to other modules of the application.

To identify to angular that there is a component, we use the @Component decorator and to be able to use it

**5 -component life cycle**

**https://angular.io/guide/lifecycle-hooks**

Lifecycle sequence

*After* creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

|  |  |
| --- | --- |
| **Hook** | **Purpose and Timing** |
| ngOnChanges() | Respond when Angular (re)sets data-bound input properties. The method receives a [SimpleChanges](https://angular.io/api/core/SimpleChanges) object of current and previous property values.  Called before ngOnInit() and whenever one or more data-bound input properties change. |
| ngOnInit() | Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.  Called *once*, after the *first* ngOnChanges(). |
| ngDoCheck() | Detect and act upon changes that Angular can't or won't detect on its own.  Called during every change detection run, immediately after ngOnChanges() and ngOnInit(). |
| ngAfterContentInit() | Respond after Angular projects external content into the component's view / the view that a directive is in.  Called *once* after the first ngDoCheck(). |
| ngAfterContentChecked() | Respond after Angular checks the content projected into the directive/component.  Called after the ngAfterContentInit() and every subsequent ngDoCheck(). |
| ngAfterViewInit() | Respond after Angular initializes the component's views and child views / the view that a directive is in.  Called *once* after the first ngAfterContentChecked(). |
| ngAfterViewChecked() | Respond after Angular checks the component's views and child views / the view that a directive is in.  Called after the ngAfterViewInit and every subsequent ngAfterContentChecked(). |
| ngOnDestroy() | Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.  Called *just before* Angular destroys the directive/component. |

**6 - what @Input and @Output**@Input allows you to pass data into your controller and templates through html and defining custom properties. This allows you to easily reuse components and have them display different values for each instance of the renderer.  
  
 **@Outputs:** Components push out events using a combination of an @Output and an EventEmitter. This allows a clean separation between reusable Components and application logic.

**7 - What is @**

**Is a alias only**

## **8 - how ngChange will work with example OnChanges()**

Angular calls its ngOnChanges() method whenever it detects changes to input properties of the component (or directive). This example monitors the [OnChanges](https://angular.io/api/core/OnChanges) hook.

on-changes.component.ts (excerpt)

content\_copyngOnChanges(changes: [SimpleChanges](https://angular.io/api/core/SimpleChanges)) {

for (let propName in changes) {

let chng = changes[propName];

let cur = JSON.stringify(chng.currentValue);

let prev = JSON.stringify(chng.previousValue);

this.changeLog.push(`${propName}: currentValue = ${cur}, previousValue = ${prev}`);

}

}

The ngOnChanges() method takes an object that maps each changed property name to a [SimpleChange](https://angular.io/api/core/SimpleChange) object holding the current and previous property values. This hook iterates over the changed properties and logs them.

The ngOnChanges() method is your first opportunity to access those properties. Angular calls ngOnChanges() before ngOnInit() and many times after that. It only calls ngOnInit() once.

**9 - what is subjective observable**

**10 - what is subscribe**The observable subscribe method is used to subscribe to messages that are sent to an observable.

**11 - what is service**

**12 - how to share data between components**

**@input – via services**

**13.**Metadata Properties:

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

**JavaScrip Question -**

**what is closure with example** function person(pname){  
 this.name=pname;  
 var getName= function(){  
 return name;  
 };  
}

var me = new person(“Ashwini”);  
  
o/p me.getName()

**what is diff let, const and var**[**https://dzone.com/articles/javascript-difference-between-var-let-and-const-ke**](https://dzone.com/articles/javascript-difference-between-var-let-and-const-ke)

## **Var**

The JavaScript variables statement is used to declare a variable and, optionally, we can initialize the value of that variable.

Example: var a =10;

* Variable declarations are processed before the execution of the code.
* The scope of a JavaScript variable declared with var is its current execution context.
* The scope of a JavaScript variable declared outside the function is global.

Consider the following code snippet.

function nodeSimplified(){

var a =10;

console.log(a); // output 10

if(true){

var a=20;

console.log(a); // output 20

}

console.log(a); // output 20

}

In the above code, you can find, when the variable is updated inside the if loop, that the value of variable "a" updated 20 globally, hence outside the if loop the value persists. It is similar to the Global variable present in other languages. But, be sure to use this functionality with great care because there is the possibility of overriding an existing value.

## **let**

The **let** statement declares a local variable in a block scope. It is similar to **var**,inthat we can optionally initialize the variable.

Example: let a =10;

* The let statement allows you to create a variable with the scope limited to the block on which it is used.
* It is similar to the variable we declare in other languages like Java, .NET, etc.

Consider the following code snippet.

function nodeSimplified(){

let a =10;

console.log(a); // output 10

if(true){

let a=20;

console.log(a); // output 20

}

console.log(a); // output 10

}

It is almost the same behavior we see in most language.

function nodeSimplified(){

let a =10;

let a =20; //throws syntax error

console.log(a);

}

**Error Message: Uncaught SyntaxError: Identifier 'a' has already been declared.**

However, with var, it works fine.

function nodeSimplified(){

var a =10;

var a =20;

console.log(a); //output 20

}

The scope will be well maintained with a let statement and when using an inner function the let statement makes your code clean and clear.

I hope the above examples will help you better understand the var and let commands and if you have any queries please write me in the comment section.

## **const**

const statement values can be assigned once and they cannot be reassigned. The scope of const statement works similar to let statements.

Example: const a =10;

function nodeSimplified(){

const MY\_VARIABLE =10;

console.log(MY\_VARIABLE); //output 10

}

As per usual, naming standards dictated that we declare the const variable in capital letters. **const a =10**will work the same way as the code given above. Naming standards should be followed to maintain the code for the long run.

**Question: What will happen when we try to reassign the const variable?**  
Consider the following code snippet.

function nodeSimplified(){

const MY\_VARIABLE =10;

console.log(MY\_VARIABLE); //output 10

MY\_VARIABLE =20; //throws type error

console.log(MY\_VARIABLE);

}

**Error Message : Uncaught TypeError: Assignment to constant variable.**

The code will throw an error when we try to reassign the existing const variable.

**what is arrow function with example**

Using arrows functions in ES6 allows us to stop using that = this or self = this or \_this = this or .bind(this). For example, this code in ES5 is ugly:

var \_this = this

$('.btn').click(function(event){

\_this.sendData()

})

This is the ES6 code without \_this = this:

$('.btn').click((event) =>{

this.sendData()

})

Sadly, the ES6 committee decided that having skinny arrows is too much of a good thing for us and they left us with a verbose old functioninstead.

**what is ES6 and what new features in that**

[**https://webapplog.com/es6/**](https://webapplog.com/es6/)

**arrow function**

**---------------**

**what is scope in javascript  
JavaScript** has two scopes: global and local. A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout your program. A variable that is declared inside a function definition is local

**javascript design pattern**

[**https://dzone.com/articles/design-pattern-in-javascript-part-1**](https://dzone.com/articles/design-pattern-in-javascript-part-1)

**HTML & CSS question**

**what is diff between html5 and html4**

* New parsing rules oriented towards flexible parsing and compatibility
* New elements – section, video, progress, nav, meter, time, aside, canvas
* New input attributes – dates and times, email, url
* New attributes – ping, charset, async
* Global attributes (that can be applied for every element) – id, tabindex, repeat
* Deprecated elements dropped – center, font, strike

**css3 box model**

**content padding border margin**

**what are the position property like - relative, absolute**

**what is bootstrap**