**Chapter: 5. Forms implementation, RxJS, Observables introductions**

**Objective:** In this chapter we are going to study below topics:

* Introduction to Forms
  + Template Driven and Model Driven Forms
  + Validations
* Introduction to RxJS
* RxJS Observables, Promices
* RxJS Operators (Table and Clear Discription)
* HttpRequest using InMemory angular web api (for reference example:
* Get Request
* Post Request
* Put Request
* Transform Responses Easily with Observable Operators (map())
* Catching Http Errors
* Multiple Choice Questions
* FAQ/Lab assignments
* Summery about the chapter.

# Angular Forms:

1. Form is a container that comprises of set of elements, which allow interaction with our application.
2. Form provides an UI from where user can input, edit, delete, view data.
3. HTML form comprises of elements like button, textbox, checkbox, radio, listbox etc.
4. Angular makes the static HTML form into dynamic.
5. HTML presents the form and Angular makes it interactive to handle client-side interactions.
6. Based on where Angular is handling interactions the forms in angular are classified into 2 types
   * Template Driven Forms
   * Model Driven Form / Reactive Form

**Template Driven Forms:**

1. A template driven form configures and handles all interactions at View Level (HTML)
2. Configuration of a form and its manipulation both handled in HTML template.
3. Very optimized controller level interaction. All interactions are at view level.
4. It reduces the number of requests to a component.
5. It improves the page load time.It is good for forms designed in “in-line” technique.
6. Template drive form is heavy on page. Slow in handling interactions and rendering. Hard to test and extend the form.
7. Separation issues. Not loosely coupled.
8. You can use template driven forms when you are designing an UI that doesn’t require regular extensions.
9. The directives that are used to configure “Form and Form Elements” in template driven approach.
   * NgForm
   * NgModel

**NgForm**: It provides a set of properties and methods that are used to configure and handle <form> element.

**NgModel**: It provides a set of properties and methods that are used to configure and handle a form control like button, textbox, checkbox, radio, dropdown list, etc.

* The library for “NgForm and NgModel” is “@angular/forms”
* The module is “FormsModule”.

**Form State Validation Services:** Form state validation services verifies all fields in the form simultaneously at the same time, Angular verifies all fields in the forms before submitting and report errors.

|  |  |  |
| --- | --- | --- |
| **Service Name** | Property | Description |
| **NgPristine** | pristine | * It returns Boolean true when form is untouched. * All fields loaded but no modification identified. |
| **NgDirty** | dirty | * It returns Boolean true when form is modified. * At least one field in the form modified then entire form is recognized as dirty. |
| **NgValid** | valid | * It returns true when all fields in the form are in valid state. |
| **NgInvalid** | invalid | * It returns true when any one form field state is recognized as invalid. |
| **NgSubmitted** | submitted | * It returns true on form submit. |

**Note**: All angular validation services return Boolean value

**Syntax**:

formName.propertyName

frmRegister.invalid

frmRegister.pristine

**Example with Template Driven Forms:**

Step-1: Create a component with the name TemplateDemo and write the following code

**templatedemo.component.html:**

<div class="container-fluid">

    <form #frmRegister="ngForm">

      <dl>

          <h2>Register User</h2>

          <dt>User Name</dt>

          <dd>

              <input type="text" name="txtName" ngModel #txtName="ngModel" class="form-control" required>

              <span \*ngIf="frmRegister.submitted && txtName.invalid || txtName.touched && txtName.invalid" class="text-danger">Name Required</span>

          </dd>

          <dt>Mobile</dt>

          <dd>

              <input type="text" name="txtMobile" ngModel #txtMobile="ngModel" class="form-control" required>

              <span \*ngIf="frmRegister.submitted && txtMobile.invalid || txtMobile.touched && txtMobile.invalid" class="text-danger">Mobile Required</span>

<br>

<input type="checkbox" name="chkRemeber" id="chkRemeber"> Remember Me

          </dd>

          <button class="btn btn-primary btn-block">Register</button>

      </dl>

    </form>

  </div>

**Templatedemo.component.ts:**

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

@Component({

  selector: 'app-templateform',

  templateUrl: './templateform.component.html',

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

})

export class TemplateformComponent {

   public register;

   public onFormSubmit(obj){

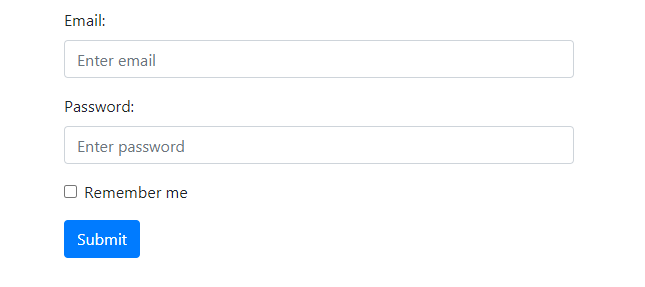
      this.register = obj;

      alert(this.register.txtName);

   }

}

**Output:**



**Reactive Forms or Model Driven Forms**

1. Reactive forms provide Model Driven approach.
2. They are bound to model so that any change in model will update the view.
3. A model driven approach binds the view with data structure.Configuration of forms and controls are defined at application logic level. (controller)
4. Easy to extend and loosely coupled.
5. Easy to test. Clean separation of functionality and presentation (Implementation and Design)
6. Reactive forms are asynchronous, they allow to submit only a specific portion of form. Type support partial updates.
7. You can dynamically add or remove controls from form.
8. The library required for configuration and implementation of reactive forms is   
   “@angular/forms”
9. The classes required for configure forms and controls dynamically
   * ReactiveFormsModule
   * FormsModule

**Example with Model Driven/Reactive forms:**

Create a component with the name **ModelDriven** and write the following code

**Model-driven.component.ts:**

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

import { FormControl, FormGroup, Validators } from '@angular/forms';

@Component({

  selector: 'model',

  templateUrl: './model-driven.component.html',

  styleUrls: ['./model-driven.component.css']

})

export class ModelDrivenComponent  {

  msg:string="";

  myform:FormGroup;

  constructor() {

    this.myform= new FormGroup({

      firstname:new FormControl("",[Validators.required,Validators.minLength(3),Validators.maxLength(20),Validators.pattern("^[a-zA-Z]\*$")]),

      lastname:new FormControl("",[Validators.required,Validators.minLength(3),Validators.maxLength(20),Validators.pattern("^[a-zA-Z]\*$")]),

      email:new FormControl("",[Validators.required,Validators.pattern("^[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$")]),

      amount:new FormControl("",[Validators.required]),

      gender:new FormControl("",[Validators.required]),

      country:new FormControl("",[Validators.required])

    })

   }

   onRegisterClick(){

     if(this.myform.valid){

       //console.log('Form Submitted..!');

       this.msg = "Data submitted Successfully";

     }

     else{

       //console.log('Invalid data..!');

       this.msg = "Require validations..!";

     }

   }

}

**Model-driven.component.html:**

<div>

    <h4>Reactive/Model Driven Forms</h4>

    <form [formGroup]="myform">

        Firstname:

        <input type="text" formControlName="firstname">

        <span class="error" \*ngIf="myform.controls.firstname.touched && myform.controls.firstname.invalid && myform.controls.firstname.errors.required">First Name Mandatory</span>

        <span class="error" \*ngIf="myform.controls.firstname.touched && myform.controls.firstname.invalid && myform.controls.firstname.errors.minlength">Min 5 Charectres</span>

        <span class="error" \*ngIf="myform.controls.firstname.touched && myform.controls.firstname.invalid && myform.controls.firstname.errors.maxlength">Max 20 Charectres</span>

        <span class="error" \*ngIf="myform.controls.firstname.touched && myform.controls.firstname.invalid && myform.controls.firstname.errors.pattern">Should allow only alphabets</span>

        <br>

        Lastname:

        <input type="text" formControlName="lastname">

        <br>

        Email:

        <input type="text" formControlName="email">

        <br>

        Amount:

        <input type="text" formControlName="amount">

        <br>

        Gender:

        <input type="radio" formControlName="gender" value="Male"><span>Male</span>

        <input type="radio" formControlName="gender" value="Female"><span>Female</span>

        <br>

        Country:

        <select formControlName="country">

            <option>Please Select</option>

            <option>India</option>

            <option>USA</option>

            <option>UK</option>

            <option>Japan</option>

        </select>

        <br>

        <input type="submit" value="Submit" (click)="onRegisterClick()">

        <br>

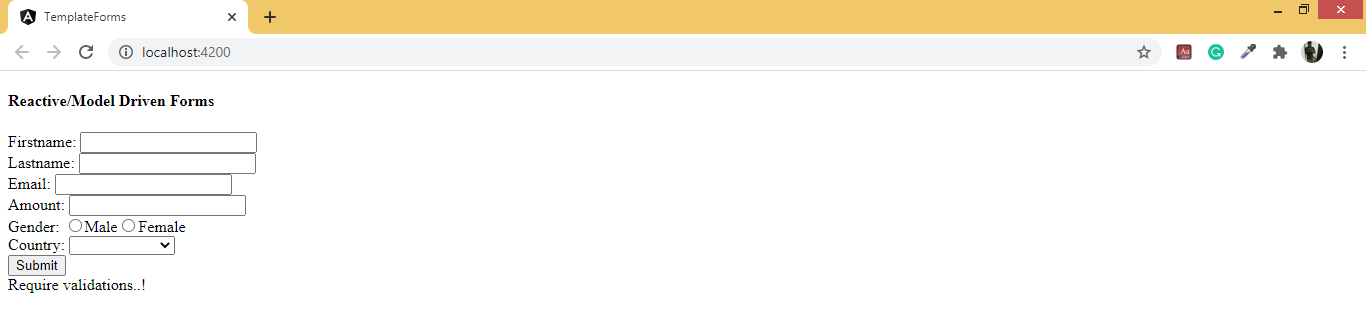
        <div [innerHTML]="msg"></div>

    </form>

</div>

**Note:** Bootstrap the component and run it will get the following

**output**



# Introduction to RxJS:

1. RxJs stands for Reactive X JavaScript programming.
2. The RxJx is actually a Js library with comes with Angular framework, and is meant for helping in asynchronous programming.
3. It is particularly known for observables and a huge collection of RxJx operators, which we will study in this chapter.

**Note:** Asynchronous programming means one call has not to wait for the completion of previous call.

**Promises vs observables**

* Promises handle only async requests, and Observable also does the same. However, do they vary?
* You should know that Observable is lazy though the promises are not.
* This is because we have to subscribe to the observables for getting the result. However, in the case of promises, the execution is automatic.
* Observable can handle multiple values however, promises cannot.
* Promises can merely provide one value whereas the observables can provide you multiple values.
* You can cancel the observables by unsubscribing it with the help of the unsubscribe through promises that do not have this feature.
* Observable comes with many operators like map, forEach, filter, etc.
* However, promises do not provide these

These are all the features that make observable differences from promises.

**Observers and Subscriptions**

You will find these important while you apply the observable. The observers happen to be the listeners or the consumers who listen or subscribe to the observed data.

**From the RxJS docs:**

The collection of callbacks is the observer that can understand how it should listen to the values that are being delivered by the Observable. The subscription happens to be the objects which are being sent back or returned when someone subscribes for the observable. You will find then available with loads of methods like the unsubscribing () method that you can call when you want to unsubscribe from receiving the observable published values.

**From the Official Docs:**

The subscription represents the execution of the Observable and is mainly important for canceling the execution.

**Subject in RxJS**

It’s a unique kind of observable and which the observer can also subscribe for fetching the published values automatically when they are published though there is one dissimilarity. The value is being multicasted to a lot of observers.

**Note:** RxJs Observable by default are unicast.

And this means that each subscribed observer is going to execute independently the observable whereas the multicast means that execution is shared. And the subjects are the same as Event Emitters.

And hence when we make use of subjects and not plain observables. All of the subscribed observersare going to get the same emitted value data.

Note: Subjects are also called Observers and that is they can subscribe to other observable and listen to the published data.

# RxJS Operators

Below are the observables list:

* Creation
* Join
* Transformation
* Mathematical
* Conditional
* Multicasting
* Utility
* Error handling
* Filtering

And lets now discuss various examples of each of the above.

**Creation Operators**

Following are the operators we are going to discuss in Creation operator category −

|  |  |
| --- | --- |
| Sr.No | Operator &Details |
| 1 | [ajax](https://www.tutorialspoint.com/rxjs/rxjs_operator_create_ajax.htm)  The ajax operator is going to make the ajax request for the given URL. |
| 2 | [from](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_from.htm)  This operator creates from an array an observable, an object of type of array, a promise, object of observable type, and an iterable object. |
| 3 | [fromEvent](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_fromevent.htm)  This gives the output in the form of an observable which is to be applied on elements that emits an event like clicks, buttons etc. |
| 4 | [fromEventPattern](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_fromeventpattern.htm)  This operator creates an observable given input function which is used for registering the event handlers. |
| 5 | [interval](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_interval.htm)  This creates an observable for each time for the given time. |
| 6 | [of](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_of.htm)  This operator is going to take in the arguments being passed and it turns them to observable. |
| 7 | [range](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_range.htm)  The operator creates an observable which gives the sequence of numbers that is based on the range given. |
| 8 | [throwError](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_throwerror.htm)  This operator creates an observable which notifies the error. |
| 9 | [timer](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_timer.htm)  This operator makes an observable which will emit the value following timeout and the value keeps incrementing with each call. |
| 10 | [iif](https://www.tutorialspoint.com/rxjs/rxjs_create_operator_iif.htm)  This operator decides which of the Observable is going to be subscribed. |

**Join Operators**

The following operators comes in Join Operator category

|  |  |
| --- | --- |
| Sr.No | Operator &Details |
| 1 | [concat](https://www.tutorialspoint.com/rxjs/rxjs_join_operator_concat.htm)  This sequentially emit the observable mentioned as input and follows with next one. |
| 2 | [forkJoin](https://www.tutorialspoint.com/rxjs/rxjs_join_operator_forkjoin.htm)  This operator is taken in an array or the dict object as input and waits for the observable to complete and returns the last values that is emitted from the provided observable. |
| 3 | [merge](https://www.tutorialspoint.com/rxjs/rxjs_join_operator_merge.htm)  This takes in the input observable and emits all the values from the observable and it emits one observable as single output. |
| 4 | [race](https://www.tutorialspoint.com/rxjs/rxjs_join_operator_race.htm)  It gives back an observable which is the mirror copy of the first source observable. |

**Transformation Operators**

The below are the operators under Transformation operator category.

|  |  |
| --- | --- |
| Sr.No | Operator & Details |
| 1 | [buffer](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_buffer.htm)  The buffer works on an observable and it considers argument as an observable. It starts to buffer the values that are emitted on the original observable in an array and will emit it when the observable considered as argument, emits. And once the observable considered as arguments emits, the buffer s reset and it starts the buffering again on original until input observable stops emitting and same process is repeated, |
| 2 | [bufferCount](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_buffercount.htm)  the buffercount() operator collects the values from the input observable on which it is called and it emits the same if the buffer size provided to buffercount is matched. |
| 3 | [bufferTime](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_buffertime.htm)  This is same as bufferCount, and it collects the values from the input obersable on which it is invoked and it emits the bufferTimeSpan. It calls for observable i.e, bufferTimeSpan. |
| 4 | [bufferToggle](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_buffertoggle.htm)  The bufferToggle(0 takes 2 arguments,closingSelector and opening. The opening arguments are subscribableor a promise for starting the buffer and the closingSelector is again subscribable or promise an indicator for closing the buffer and it emits the values collected. |
| 5 | [bufferWhen](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_bufferwhen.htm)  This will give the values in the array form, it awaits one argument in the form of function which will decide when the buffer should be reset, emit or closed. |
| 6 | [expand](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_expand.htm)  The expand operator accepts a function as an argument, that is applied on the source observable recursively and also on the output observable. And observable is the final value. |
| 7 | [groupBy](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_groupby.htm)  In this, the output is grouped and grouping is based on the specific condition and these group items are being emitted as the GroupedObservable. |
| 8 | [map](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_map.htm)  In case of map operator, the project function is being applied on each value on the source Observable and Observable emits the same output. |
| 9 | [mapTo](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_mapto.htm)  The constant value is given as output together with the Observable each time the value is emitted by the source Observable. |
| 10 | [mergeMap](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_mergemap.htm)  In this case, the project function is applied on every source value, and the output of this is merged with output observable. |
| 11 | [switchMap](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_switchmap.htm)  In this case, the project function is being applied on each of the source value, and its output of it is merged with the output Observable, and the value provided is projected Observable and the most recent one. |
| 12 | [window](https://www.tutorialspoint.com/rxjs/rxjs_transformation_operator_window.htm)  It tales windowsboundaries as argument, that is an observable, and it provides back a nested observable each time the windowboundaries emits. |

**Mathematical Operators**

The following operators are being considered as the Mathematical operators.

|  |  |
| --- | --- |
| Sr.No | Operator & Details |
| 1 | [Count](https://www.tutorialspoint.com/rxjs/rxjs_mathematical_operator_count.htm)  The count() operator takes in an Observable with values and converts it into an Observable that will give a single value |
| 2 | [Max](https://www.tutorialspoint.com/rxjs/rxjs_mathematical_operator_max.htm)  Max method will take in an observable with all values and return an observable with the max value |
| 3 | [Min](https://www.tutorialspoint.com/rxjs/rxjs_mathematical_operator_min.htm)  Min method will take in an observable with all values and return an observable with the min value. |
| 4 | [Reduce](https://www.tutorialspoint.com/rxjs/rxjs_mathematical_operator_reduce.htm)  In reduce operator, accumulator function is used on the input observable, and the accumulator function will return the accumulated value in the form of an observable, with an optional seed value passed to the accumulator function.  The reduce() function will take in 2 arguments, one accumulator function, and second the seed value. |

**Conditional Operators**

The following are the operators we are going to discuss in the conditional operator category.

|  |  |
| --- | --- |
| Sr.No | Operator & Details |
| 1 | [defaultIfEmpty](https://www.tutorialspoint.com/rxjs/rxjs_conditional_operator_defaultifempty.htm)  The operator is going to return the default value if the source observable is empty. |
| 2 | [every](https://www.tutorialspoint.com/rxjs/rxjs_conditional_operator_every.htm)  This is going to return an observable which is dependent on the input function that satisfies the condition on every value on the source Observable, |
| 3 | [find](https://www.tutorialspoint.com/rxjs/rxjs_conditional_operator_find.htm)  This is going to return the observable when the first value of the source Observable satisfies the condition for the predicate function which is taken as the input. |
| 4 | [findIndex](https://www.tutorialspoint.com/rxjs/rxjs_conditional_operator_findindex.htm)  This operator is based on the input scheduler and it will reemit the notification from the source Observable. |
| 5 | [isEmpty](https://www.tutorialspoint.com/rxjs/rxjs_conditional_operator_isempty.htm)  this gives the output as true if the input observable looks for complete callback without emitting any values, and false if the input observable emits even at least a value. |

**Multicasting Operators**

The following operators comes under the list of multicasting operator.

|  |  |
| --- | --- |
| Sr.No | Operator & Details |
| 1 | [multicast](https://www.tutorialspoint.com/rxjs/rxjs_multicasting_operator.htm)  These shares the single subscription being created with other subscribers. The params which multicast accepts, is a subject or a factory method which returns a ConnectableObservable which has connect() method. |
| 2 | [publish](https://www.tutorialspoint.com/rxjs/rxjs_multicasting_operator_publish.htm)  This operator returns back ConnectableObservable and it requires using the connect() method for subscribing to the Observable |
| 3 | [publishBehavior](https://www.tutorialspoint.com/rxjs/rxjs_multicasting_operator_publishbehavior.htm)  publishBehavior makes use of BehaviorSubject and it returns ConnectableObservable. The connect() method is to be used for subscribing to the observable that is created. |
| 4 | [publishLast](https://www.tutorialspoint.com/rxjs/rxjs_multicasting_operator_publishlast.htm)  The publishBehavior makes the use of the AsyncSubject and it gives back the ConnectableObservable. The connect() method is to be used for subscribing to the observable created. |
| 5 | [publishReplay](https://www.tutorialspoint.com/rxjs/rxjs_multicasting_operator_publishreplay.htm)  The publishReplay uses the behavior subject that in which it can buffer the values and it replays that to the new subscribers and it returns ConnectableObservable. The connect() method is to be used for subscribing to the observable that is created. |
| 6 | [share](https://www.tutorialspoint.com/rxjs/rxjs_multicasting_operator_share.htm)  this is multicast() alias operator and the only difference is that you are not required to call the connect(0 method manually for starting the subscription. |

**Utility Operators**

The below are the operators that comes under the utility operator list.

|  |  |
| --- | --- |
| Sr.No | Operator & Details |
| 1 | [tap](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_tap.htm)  This have the output which is same as the source observable, and it can be used to log the values to the user from the observable. And the main value, if an error or on completion of the task. |
| 2 | [delay](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_delay.htm)  This delays the values that are emitted from the source Observable depending on the given timeout. |
| 3 | [delayWhen](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_delaywhen.htm)  This operator delays the emitting of values from the source Observable depending on the timeout from another observable considered as input. |
| 4 | [observeOn](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_observeon.htm)  The operator is grounded on the input scheduler will reemit the notification as of the source observable |
| 5 | [subscribeOn](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_subscribeon.htm)  This operator leverages the developer to subscribe asynchronously to the source Observable depending on the scheduler considered as input. |
| 6 | [timeInterval](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_timeinterval.htm)  This operator returns an object that contains the current value and the time taken between the current and the previous value which is calculated with the help of the scheduler input taken. |
| 7 | [timestamp](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_timestamp.htm)  This returns the timestamp together with the value that is emitted from the source Observable that tells about the time the value is emitted. |
| 8 | [timeout](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_timeout.htm)  This operator throws an error if the source Observable do not emit the value after the given timeout. |
| 9 | [toArray](https://www.tutorialspoint.com/rxjs/rxjs_utility_operator_toarray.htm)  collects all the source value from the Observable and it outputs them in the form of array when the source completes. |

**Error Handling Operators**

These are the operators considered as error handling operator.

|  |  |
| --- | --- |
| Sr.No | Operator & Details |
| 1 | [catchError](https://www.tutorialspoint.com/rxjs/rxjs_error_handling_operator_catcherror.htm)  This operator catches the errors on the source Observable by returning an error or a new Observable. |
| 2 | [retry](https://www.tutorialspoint.com/rxjs/rxjs_error_handling_operator_retry.htm)  This operator takes care of the retrying again on the soirce observable in case of the error and the retry is to be done depending on the input count given. |

**Filtering Operators**

The following operators comes under Filtering Operators:

|  |  |
| --- | --- |
| Sr.No | Operator & Details |
| 1 | [debounce](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_debounce.htm)  The value emitted from the source Observable after few seconds and the emission is being determined by next input provided as Observable pr promise. |
| 2 | [debounceTime](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_debouncetime.htm)  This will emit value from the source observable however after the time completes only. |
| 3 | [distinct](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_distinct.htm)  This operator is providing all the values from the source observable which are distinct when it is compared with the previous value. |
| 4 | [elementAt](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_elementat.htm)  This operator provides a single value from the source observable depending on the index provided. |
| 5 | [filter](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_filter.htm)  The operator filter values from the source Observable depending on the predicate function provided. |
| 6 | [first](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_first.htm)  This operator provides the first value which is emitted from the source Observable. |
| 7 | [last](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_last.htm)  This operator provides the last value that is emitted by the source Observable. |
| 8 | [ignoreElements](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_ignoreelements.htm)  This ignores the values from the source Observable and it executes the call only to complete or error callback functions. |
| 9 | [sample](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_sample.htm)  This operator provides the most recent value from the source Observable, and the output depends upon the argument that provided to it emits. |
| 10 | [skip](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_skip.htm)  This operator returns an observable which skips the first occurrence of the count items accepted as input. |
| 11 | [throttle](https://www.tutorialspoint.com/rxjs/rxjs_filtering_operator_throttle.htm)  This will output and ignore the values from the source observable for the time determined by the input function accepted as an argument and the same process is repeated. |

# Example using RxJS and HTTP Requestsusing JSON—server:

The Angular provides you with the HTTP option for communicating with the server and fetching the data from the server. The majority of the front-end applications communicate with the server via the HTTP protocol for downloading and uploading the data and accessing various other back end services. And Angular provides you with the simplified HTTP API for the Angular application, the **HTTPClient** service class which you can find in

**@Angular/common/HTTP**

* The HTTP client service does offer the below major features.
* The ability for requesting the typed response objects.
* Better streamlined error handling
* Better features for testability
* Request and response interception.

**Prerequisites**

Before you start working with the **HTTPClientModule**, you need to know the following:

* TypeScript
* HTTP protocol usage
* Angular app design fundamental
* The observable techniques as well as operators.

And it all starts with the:

**Import {HttpClientModule} from ‘@angular. /common/http’;**

All the apps import the above. And you are importing the module in the app.module.ts.

Now as you have the **HTTPClientModule** included, you can inject the HTTPClient service in the app.component.ts file or any of the components or service. And you can do that with below command:

Import {HTTPClient} from ‘@angular/common/http’;

And you need to import as well

Import {browser/module} from ‘@angular/platform-browser’

Also, you need to mention **BrowserModel** and **HTTPClientModule**in the import section of **@NgModule**.

The **HTTPClient** service applies the **Observables** for all the transactions. You need to import the **Observable** and **operator** symbols from the **RxJx module**.

Now for now have a look at the below code:

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

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

@Injectable({

  providedIn: 'root'

})

export class CommonService {

  constructor(private \_http:HttpClient) { }

  createUser(user){

    return this.\_http.post("http://localhost:3000/users",user);

  }

  getAllUser(){

    return this.\_http.get("http://localhost:3000/users")

  }

  updateUser(user)

  {

    return this.\_http.put("http://localhost:3000/users/"+user.id,user)

  }

  deleteUser(user){

    return this.\_http.delete("http://localhost:3000/users/" +user.id)

  }

}

This is a service and above: we are using one function for posting or creating, one for fetching in a grid through HTTP GET, one is for updating through HTTP PUT and one is for deleting using http delete.

**HTTPCLIENT Example using JSON server.**

For testing, we can also make use of the Json server. And we will be using that here. So our first objective is to create an app, and install the json server inside it in the server folder. The command for creation of app is known to you, and the one used for JSON server is as below:

* Move inside app folder
* mkdir server
* cd server
* npm install -g json-server
* Create a db.json file with some data

Add this data to **db.json**

{

    "users":[

      {

        "name": "Saurabh Khare",

        "Phone": "2622222",

        "email": "saurabh@nareshit.com",

        "password": "552528777",

        "id": 13

      },

      {

        "name": "sam",

        "Phone": "2622222",

        "email": "saurabh@nareshit.com",

        "password": "46464646",

        "id": 14

      }

    ];

  }

**Now add below code to App.component.ts**

**App.component.ts:**

console.log(FormObj);

    this.commonService.createUser(FormObj).subscribe((response)=>{console.log("User has been added")})

    this.getCurrentUser();

  }

  getCurrentUser(){

    this.commonService.getAllUser().subscribe((response)=>{this.allPerson=response})

  }

  deletePerson(user){

    this.commonService.deleteUser(user).subscribe(()=>{this.getCurrentUser();})

  }

  editPerson(user){

    this.isEdit=true;

    this.personObj=user;

  }

  updatePerson(user){

    this.isEdit=!this.isEdit;

     this.commonService.updateUser(this.personObj).subscribe(()=>{this.getCurrentUser()});

  }

}

Add this code to **App.component.html**

<form #myForm="ngForm">

    <div class="form-group">

      <label >Name</label>

      <input type="text" class="form-control" name="name"  [(ngModel)]="personObj.name">

      <small id="emailHelp" class="form-text text-muted">Enter your Name</small>

    </div>

    <div class="form-group">

      <label >Phone Number</label>

      <input type="tel" class="form-control" name="Phone" [(ngModel)]="personObj.Phone">

      <small id="emailHelp" class="form-text text-muted">Enter your Phone Number</small>

    </div>

    <div class="form-group">

      <label >Email Address</label>

      <input type="email" class="form-control" name="email"  [(ngModel)]="personObj.email">

      <small id="emailHelp" class="form-text text-muted">Enter your Email Address</small>

    </div>

    <div class="form-group">

      <label >Password</label>

      <input type="password" class="form-control" name="password" [(ngModel)]="personObj.password">

      <small id="emailHelp" class="form-text text-muted">Enter your Password.</small>

    </div>

    <button type="button" class="btn btn-primary" (click)="addUser(myForm.value)" \*ngIf="!isEdit">Submit</button>

    <button type="button" class="btn btn-primary" \*ngIf="isEdit">Update</button>

  </form>

  <table class="table">

    <thead>

        <tr>

            <th scope="col">#</th>

            <th scope="col">Name</th>

            <th scope="col">Mobile</th>

            <th scope="col">Email</th>

            <th scope="col">Password</th>

            <th scope="col">Edit</th>

            <th scope="col">Delete</th>

          </tr>

        </thead>

        <tbody>

          <tr \*ngFor="let person of allPerson">

            <td>{{person.id}}</td>

            <td>{{person.name}}</td>

            <td>{{person.Phone}}</td>

            <td>{{person.email}}</td>

            <td>{{person.password}}</td>

            <td><button class="btn btn-success"(click)="editPerson(person)">Edit</button></td>

            <td><button class="btn btn-warning" (click)="deletePerson(person)">Delete</button></td>

          </tr>

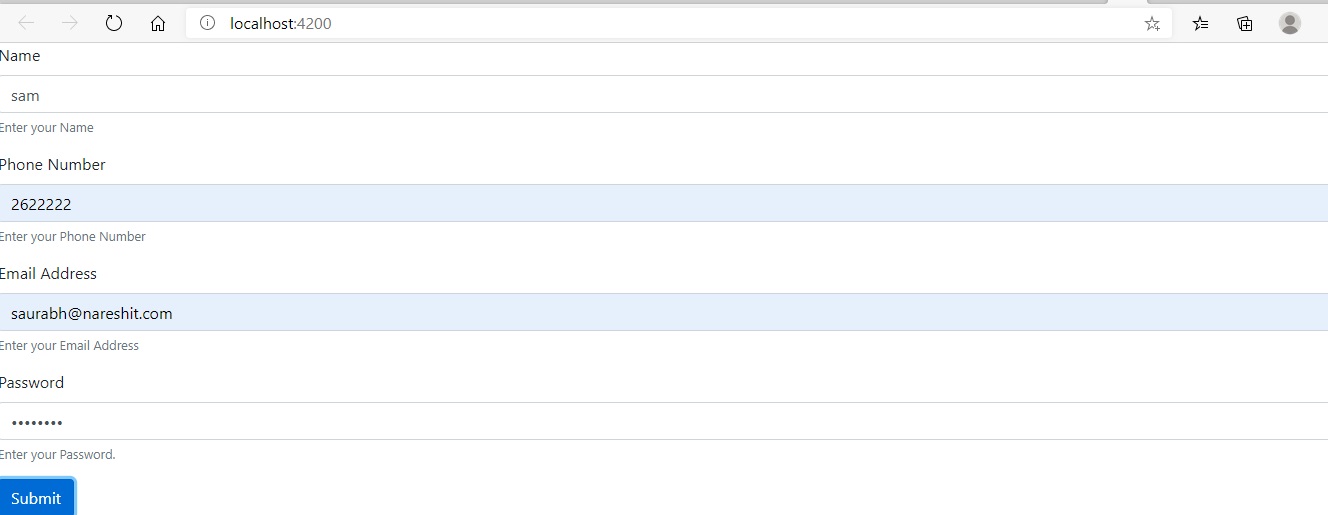
        </tbody>

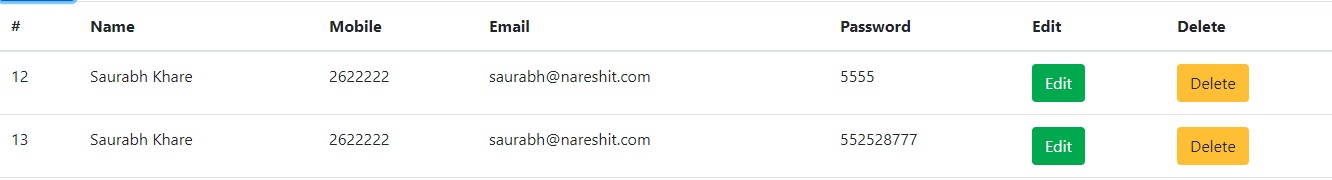
      </table>

* Add this code to the **App.module.ts**
* Add **FormsModule** in **@NgModule** and import it as well
* Import **HttpClientModule** as well, and add it in imports as well.
* Now start the **json-server** using **Json-server –watch db.json**
* And run above app using **ng serve -o**

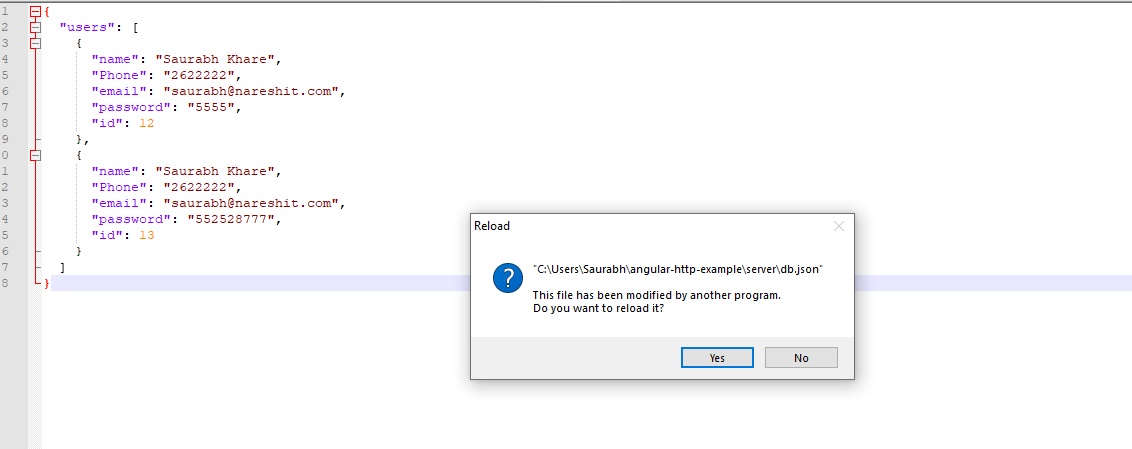
**You will get below output:**

**Form 1: for entering person data**





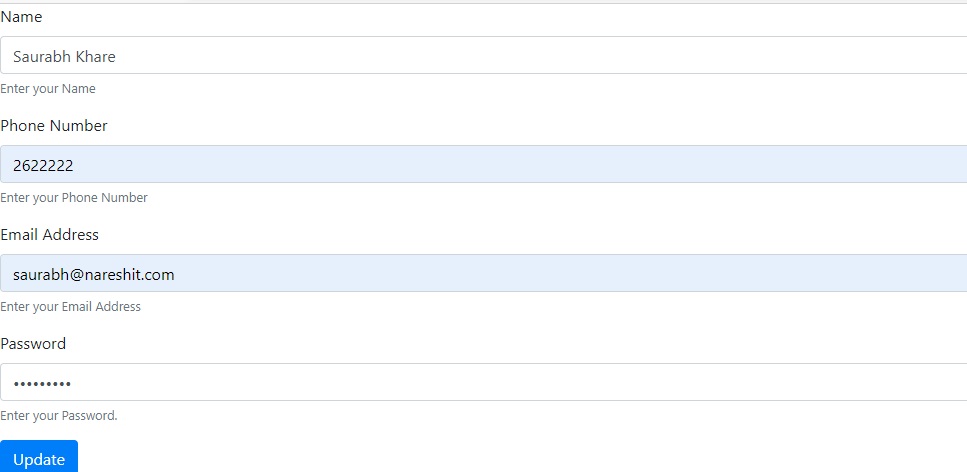
**Db.json before post**



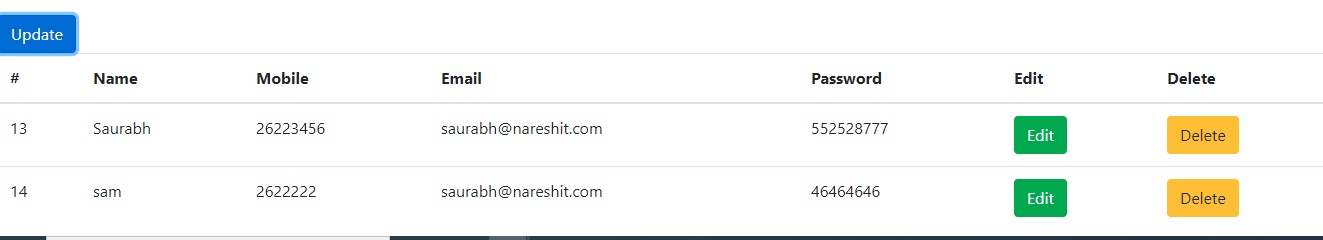
**Db.json after post**



**Dbjson before delete**



**After update:**



Thus, we have covered post, get, put as well as delete. We will be using it in Authentication and Routes as well. And below we have detailed them with some sample code.

**Adjusting Request Headers**

HTTP headers Class

This represents the header configuration options for a certain HTTP request. And its instances are immutable. All the modifying methods registers changes in cloned instances and the original objects are never changed.

class HttpHeaders {

  constructor(headers?: string | { [name: string]: string | string[]; })

  has(name: string): boolean

  get(name: string): string | null

  keys(): string[]

  getAll(name: string): string[] | null

  append(name: string, value: string | string[]): HttpHeaders

  set(name: string, value: string | string[]): HttpHeaders

  delete(name: string, value?: string | string[]): HttpHeaders

  }

**Constructor**

Constructs a new HTTP header object with the provided values.

 constructor(headers?: string | { [name: string]: string | string[]; })

Parameters are as below:

**Headers**?:string | { [name: string]: string | string[]; }

Optional, and the default is undefined.

**has():**This checks for the existence of the header.

**has(name: string):**boolean-This checks for the existence of the header name.

Returns

boolean: if the header exists then this is true or else it is false.

**get()-**the first value of the header is retrieved from this.

**get(name: string):** string | null

Parameters

name string

This is the header name

Returns

string | null: if the header exists then it is a string or else it is null.

**keys()**

This retrieves the header's name.

**keys(): string[]**

Parameters

There are parameters.

Returns

string[]: this is the list of header names.

**getAll()**

This retrieves the list of given values of the header.

**getAll(name: string)**: string[] | null

Parameters

name string

This is the header name from which the value is retrieved.

Returns

string[] | null: A string of values if the header exists or else it is null.

**append()**

This adds the fresh value to the existing value of the header and reverts as a duplicate of the original case.

**append(name: strng, val: string | string[]): HttpHeaders**

Parameters

name string

the name of the header for which to append the value.

value string | string[]

This is the value to be appended.

Returns

This returns HttpHeaders which is a clone of the HTTP headers object and the value is appended to the mentioned header.

**set()**

This sets or modifies certain values for certain headers in a clone of the original instance. And if the header already exists then its value is replaced with the provided value in the object that is returned.

**set(name: string, value: string | string[]):**HttpHeaders

Parameters

name string

This is the header name.

value string | string[]

The value or value set for setting or overriding for a header.

Returns

**HttpHeaders:** A clone of the HTTP headers with a new set of values.

**delete()**

This deletes the set of values for a given header in a clone of the original instance.

delete(name: string, value?: string | string[]): HttpHeaders

Parameters

name : string

This is the header name.

value string | string[]

The value or value set to be deleted for a given header name.

Optional. The default is undefined.

Returns

HttpHeaders: A clone of the HTTP headers object with the given object that is to be deleted.

# Multiple Choice Questions

**1. RxJs is meant for what?**

a) For multithreading b) For multiprogramming

c) For synchronous programming d) For asynchronous programming

**2. RxJs stands for**

a) Reactive JavaScript b) React JavaScript

c) Reactive Extension for JavaScript d) React XML based JavaScript

**3. Map and Merge both are operators used in RxJs**

a) Yes b) They are applied on arrays rather c) No

d) They are meant for both arrays and as RxJs operators

**4. Promise and Observable are different as**

a) Both are same b) Both are different

c) Observable can be subscribed and can have continuous returns till user

unsubscribe,

d) Promises are limit to one return

**5. Which is best out of these callbacks, promises, observables?**

a) Callback b) Promises c) Observables

**6. Http Put is same as**

a) POST b) DELETE c) UPDATE d) GET

**7. Web API are consumed by SPA using HTTP?**

a) Yes b) No c) not always

d) sometime yes and sometimes no

**8. Json server can be used for testing HTTP REQUESTS as well as in memory Web API**

a) Yes b) Only Json server

c) Only in memory web api d) None of these

**9. All rest-based APIs can be consumed by SPA?**

a) Yes b) No

c) Only Statefulapis d) State does not matter for HTTP

**10. Which app can be used to check HTTP requests?**

a) Fiddler b) Postman

c) Both a and b d) None of these

# FAQs / LAB Based Assignments

1. How can you send and request data from the server in Angular?
2. Explain briefly the App and backend setup that you need to put in place before starting angular app development.
3. How can you send request using POST request?
4. What adjustment is made in Request header while sending Get and POST request? Explain briefly.
5. How can you send GET request using HTTP Get request?
6. What is a HTTP Put request?
7. How can you transfer responses using Observables?
8. What is an Operation Heap? Explain briefly.
9. How can you catch HTTP Errirs in Angular?
10. Explain briefly the usage of async pipe using HTTP request?

# Summery

# 

# We covered following in this chapter

1. Forms implementation in Angular
2. Template Driven forms and Model Driven forms
3. We learned what is RxJs?
4. We learned what is Asynchronous programming?
5. We learned how we can code asynchronously using RxJs.
6. We learned callbacks, promises and Observables
7. We learned the advantages of Observables and how it can return many times and till the user unsubscribe.
8. We learned RxJs operators.
9. We learned how to subscribe and unsubscribe to HTTP.
10. We learned how to implement HTTP GET, POST, DELETE and UPDATE using HTTPClient.
11. And before that we learned the perquisite for Http.
12. We also saw one HTTP request example through JSON server.

And that concluded this chapter.