JSONP:

JSONP is a rest service we can use to bypass cross-origin domain restrictions, and is often used when an application needs data from a publicly available third-party API and there's no established access token. JSONP stands for JSON with Padding, and just like a GET request, it returns a response object. The difference is that JSONP returns the object inside a function with a known variable name. The function is defined and invoked locally by the application with the response object from the server passed in as an argument.

General GET request will be like:

```
{
        "foo":"bar",
        "foo":"bar"
}
```

Jsonp will be like:

```
callback({
        "foo":"bar",
        "foo":"bar"
})
```

what's an Obsvervable and what is an observer? What's the difference?

An Obsvervable is just a function. It takes in an observer and provides us with lots of methods that do different things, like create and subscribe. So what is observer mean? The observer is just an object. It has three methods, next, error and complete. The observer is what you will use to listen to whatever it is you want to listen to.

```
import { Observable } from 'rxjs';
import { Component,OnInit, OnDestroy } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
 title = 'app';
 observable$;
 ngOnInit(){
   this.observable$ = Observable.create((observer)=>{
```

```
      observer.next(1);
      observer.next(2);
      observer.next(3);
      observer.complete();
    });
    this.observable$.subscribe(
      value=>console.log(value),
      err =>{},
      ()=>console.log("this is end")
    )
  }
  ngOnDestroy(){
    //why we are doing this mean when ever the subscriotion is open for every
time. the memory leakes will happen to avoid that we will unsubscribe when ever
you leave the component
    this.observable$.unsubscribe();//unsubscribe this when ever you leave the
component
  }
}
```

Observers contain three methods: next, error, and complete.Observables are the outer wrapper that allow you to listen to an observer. One important thing to note is that each time you create an observable, you get a single instance. This means you cannot reuse or share observers. Each new observable will contain its own observerand these observers cannot interact with each other. You can subscribe to the same observable more than once but each subscription will be a separate instance and they won't know about each other.

If you call observer.next(4); outside it will not accept. so if you want to do that we have a concept called "subject"

```
import { Observable,Subject } from 'rxjs';
import { Component,OnInit, OnDestroy } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';
  mySubject$;
  ngOnInit(){
    this.mySubject$ = new Subject();
    this.mySubject$.subscribe(x=>console.log("first subscription",x));
    this.mySubject$.next(1);
```

```
    this.mySubject$.next(2);
    this.mySubject$.subscribe(x=>console.log("second subscrition",x));
    this.mySubject$.next(3);
    this.mySubject$.next(4);
  }
  ngOnDestroy(){
    this.mySubject$.unsubscribe();
  }
}
/*
second subscription done at line number 17, so the below next items will be
subscribe
we only get notified of events after we've subscribed.
output will be
first subscription 1
first subscription 2
first subscription 3
second subscrition 3
first subscription 4
second subscrition 4
What happens if you want all the values, even the ones that have already passed?
Don't worry, there's a subject for that too.
First, just one more important thing to know about subjects: subjects are
shareable but they are not reusable. Remember that once you call error or
complete on a subject, it's dead, you cannot use it again and if you try, it will
silently ignore you.
*/
```

Earlier we mentioned that there are other subjects, so let's talk about those for a minute. Remember our second subscriber?We didn't get a console log and a second subscriber until the next was called later. What if want to get the last value passed through that subject as soon as we subscribe? Then we want to use the behavior subject. The behavior subject is just like the subject, except it requires a starting value.

```
import { Observable,BehaviorSubject } from 'rxjs';
import { Component,OnInit, OnDestroy } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';
```

```
  mySubject$;
  ngOnInit(){
    this.mySubject$ = new BehaviorSubject(200);
    this.mySubject$.subscribe(x=>console.log("first subscription",x));
    this.mySubject$.next(1);
    this.mySubject$.next(2);
    this.mySubject$.subscribe(x=>console.log("second subscrition",x));
    this.mySubject$.next(3);

  }
  ngOnDestroy(){
    this.mySubject$.unsubscribe();
  }
}

/*
here first we are sending value i.e 200.
for second subscription it will take from 2nd item as argument
output will be like
first subscription 200
first subscription 1
first subscription 2
second subscrition 2
first subscription 3
second subscrition 3
*/
```

So a behavior subject is just like a subject except it requires a starting value and holds the most recent value for any new subscribers.

ReplaySubject:

Remember the second subscribe started with the number two because that was the latest value when we subscribed? What if you want to get all the values that have passed through a subject since the beginning? No problem. You just need a replay subject. The replay subject will save all the values and give them to each subscriber even after the replay subject has been completed.

```
import { Observable,ReplaySubject } from 'rxjs';
import { Component,OnInit, OnDestroy } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
```

```
  title = 'app';
  mySubject$;
  ngOnInit(){
    this.mySubject$ = new ReplaySubject();
    this.mySubject$.subscribe(x=>console.log("first subscription",x));
    this.mySubject$.next(1);
    this.mySubject$.next(2);
    this.mySubject$.subscribe(x=>console.log("second subscrition",x));
    this.mySubject$.next(3);

  }
  ngOnDestroy(){
    this.mySubject$.unsubscribe();
  }
}

/*
first subscription 1
first subscription 2
second subscrition 1
second subscrition 2
first subscription 3
second subscrition 3
*/
```

RxJs Operators:

1. Interval
2. take

```
import { Observable } from 'rxjs';
import { Component,OnInit, OnDestroy } from '@angular/core';
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';

  ngOnInit(){
```

```
    const numbers$ = interval(1000).pipe(take(5))//only 5 times this interval
will happen
    numbers$.subscribe(x=>console.log(x));
  }
  ngOnDestroy(){

  }
}
```

3. map

```
import { Observable } from 'rxjs';
import { Component,OnInit, OnDestroy } from '@angular/core';
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';

  ngOnInit(){
    const numbers$ = interval(1000)
    numbers$
    .pipe(take(5))
    .pipe(map((x)=>{
      return x*10
    }))
    .subscribe(x=>console.log(x));
  }
  ngOnDestroy(){

  }
}

```

4. Filter

```
import { Component,OnInit, OnDestroy } from '@angular/core';
import { Observable,interval } from 'rxjs';
```


Pipe is
pure function

```
import { map,filter,take } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';

  ngOnInit(){
    const numbers$ = interval(1000)
    numbers$
    .pipe(take(5))
    .pipe(map((x)=>{
      return x*10
    }))
    .pipe(filter(x => x>10))
    .subscribe(x=>console.log(x));
  }
  ngOnDestroy(){

  }
}
```

5.mergeMap:

So far, we've only had one observable at a time. What happens if we're dealing with more than one observable? Well, we have operators for that too.

```
import { Component,OnInit, OnDestroy } from '@angular/core';
import { Observable,interval,of } from 'rxjs';
import { map,filter,take,mergeMap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';

  ngOnInit(){
    const numbers$ = interval(1000)
```

of → Emit variable amount of values in sequence and then emits a complete notification.

→ Emitting a sequence of letters.

```
    const letters$ = of('a','b','c','d','e');
    letters$.pipe(mergeMap(x=>//here x will be from letters$
      numbers$
      .pipe(take(5))
      .pipe(map(y=>x+y))//y will be from numbers$
    )).subscribe((z)=>console.log(z));
  }
  ngOnDestroy(){

  }
}
```

o/p all Entries will Print

6.switchMap: Map to observable, complete previous inner observable, emit values. (cancelling Effect)

```
import { Component,OnInit, OnDestroy } from '@angular/core';
import { Observable,interval,of } from 'rxjs';
import { map,filter,take,switchMap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';

  ngOnInit(){
    const numbers$ = interval(1000)
    const letters$ = of('a','b','c','d','e');
    letters$.pipe(switchMap(x=>//recent i.e latest search will print
      numbers$
      .pipe(take(5))
      .pipe(map(y=>x+y))//y will be from numbers$
    )).subscribe((z)=>console.log(z));
  }
  ngOnDestroy(){

  }
}
```

o/p switch to latest Entries

User Interface:

Observable from a button:

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

```
import {fromEvent} from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';

  ngOnInit(){
    const source = fromEvent(document,'click');
    source.subscribe((x=>console.log(x)));
  }
  ngOnDestroy(){

  }
}
```

Debounce:

The debounce Method will wait a specific amount of time between events. For example, say you have a search bar and you want to show the user instant search results as soon as they type something, without waiting for them to hit submit.

Well that's great, however, rather than pinging your server for every single letter they type, the debounce can wait for a delay of just a few milliseconds, until we think they're done typing, then ping the server.

the debounce operator requires another observable passed in as a parameter, so I don't usually use it. Instead, I use debounceTime, because then all I have to pass in is the milliseconds, super easy. Now again, there are several ways to handle this. For me, the easiest and cleanest way is to create a subject. We'll call next on the subject and debounce it from there

```
import { Component,OnInit, OnDestroy } from '@angular/core';
import { Subject, timer } from 'rxjs';
import { debounce, debounceTime } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit,OnDestroy {
  title = 'app';
```

<input type="text" [(ngModel)]="search"
(ngModelChange)="inputChanged($event)" />

```
  serachSubject = new Subject<string>();
  ngOnInit(){

//this.serachSubject.pipe(debounceTime(200)).subscribe((x=>console.log("debounced
time",x)));
    this.serachSubject.pipe(debounce(()=>{
      return timer(500)
    })).subscribe((x=>console.log("debounced time",x)));
  }
  inputChanged($event){
    console.log("input changed",event);
    this.serachSubject.next($event)
  }
  ngOnDestroy(){

  }
}
```

<div *ngFor="let item of items | async">

what is async in above statement ?

This ngFor is a loop. So this is telling Angular to loop through each search result. And what's this? This is the AsyncPipe. This is actually quite cool. This allows us to send an observable straight to the DOM without using subscribe to unwrap it. The AsyncPipe will automatically subscribe and the AsyncPipe will automatically unsubscribe when the user leaves the page.

RxJS - Observables

Async pipe

//user.ts

```
export class User{
  fname:string;
  lname:string;
  username:string;
  password:string;
}
```

//user.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from "@angular/common/http";
import { User } from './user';

@Injectable()
export class UserService {

    constructor(private http:HttpClient) { }

    registerUser(user:User){
        return this.http.post("/api/registerUser",user);
    }
    listUsers(){
        return this.http.get("/api/listusers");
    }
    editUser(id:number){
        return this.http.get("/api/editUser/"+id);
    }
    updateUser(user:User){
        return this.http.put("/api/updateUser",user);
    }
    deleteUser(id:number){
        return this.http.delete("/api/deleteUser/"+id);
    }
    loginUser(user:User){
        return this.http.post("/api/loginUser",user);
    }
    logOut(){
      return this.http.get("/api/logout");
    }
    authStatus(){
      return this.http.get("/api/authStatus");
    }

}
```

//userList.component.ts

```
import { Observable } from 'rxjs';
import { Component } from "@angular/core";
import { UserService } from "./user.service";
import { User } from "./user";
import { Router } from "@angular/router";

@Component({
  selector: "userList",
  templateUrl: "./userList.component.html",
  styleUrls: ['./userList.component.css']
})
export class UserListComponent {
  users: Observable<User[]>;
  constructor(private cs: UserService, private router: Router) {
    this.users = <Observable<User[]>>this.cs.listUsers();

  }
  delete(userId) {
    this.cs.deleteUser(userId).subscribe((data) => {
      if (data) {
        this.users = <Observable<User[]>>this.cs.listUsers();
      }
    })
  }
  logout() {
    this.cs.logOut().subscribe((data) => {
      if (data) {
        localStorage.removeItem("user");
        this.router.navigate(['/login']);
      }
    })

  }
}
```

//userList.component.html

```
<tr *ngFor="let user of users | async">
```