RxJS   
(Reactive Extensions for JavaScript)

# Understanding RxJS

* Understanding the notion of stream of values first is the starting point before diving into understanding the notion of observables.
* Here are couple of good articles about the basics os RxJS
  + Rxjs: In 6 minutes!  
    <https://medium.com/@mohandere/rxjs-5-in-5-minutes-1c3b4ed0d8cc>
  + Learning Observable By Building Observable  
    <https://medium.com/@benlesh/learning-observable-by-building-observable-d5da57405d87>
  + Hot vs Cold Observables  
    <https://medium.com/@benlesh/hot-vs-cold-observables-f8094ed53339>

## What is stream of values?

* Stream of values is one of the core concepts of RxJS.
* As you know in a JavaScript program, everything almost is a synchronous.
* We have requests coming from the network bringing new values from the backend, and we have timeouts occurring in the front end, we have user interaction with clicks and mouse over events. All of those are synchronous events that we need to combine in order to produce the final result of our program.
* So for example every click that you do in an application that will be a **stream of values** containing the click event.
* So there are streams that emit multiple values e.g. setInverval() – some never ending, some ending based on events and also there are streams that emit only one value e.g. setTimeout()

## What is RxJS?

* In a normal JavaScript application, one of the common usercase is of combining multiple streams – e.g. on a click event, we may want to wait for 3 seconds and only then start emitting intervals.
* So with vanilla JavaScript approach, we would have to nest these 3 operations, which could lead to a big problem called – **Callback Hell** as we do deep nesting. So nesting is definitely not a convenient way from different perspectives. This is where RxJS comes to the rescue.
* RxJS stands for Reactive Extensions for JavaScript, it's a library that makes it very simple to combine streams of values together in a maintainable way.
* It's an ‘extension’ to standard JavaScript because the plain JavaScript version that we have in our browsers only has callback interfaces that really does not scale well in complexity as we start having lots of nested code. And our program starts to become harder to read and reason about.

## What is RxJS Observable?

* Observable is one of the core notions of RxJS.
* The sole purpose of Observables is to connect the observer to a producer (something that produces values), and return a means to tear down that connection to the producer. The observer is really a registry of handlers that can be pushed values over time.
* **Observables are a function that take an observer and return a function. Nothing more, nothing less.**
* **What’s a “Producer”?**
  + A producer is the source of values for your observable. It could be a web socket, it could be DOM events, it could be an iterator, or something looping over an array. Basically, it’s anything you’re using to get values and pass them to `observer.next(value)`.
* An Observable is a blueprint for stream.
* An Observable will become stream of values only if we subscribe to it. That means an observable will get values only when we explicitly subscribe to it.
  + E.g.   
    const click$ = fromEvent(document, 'click');

click$.subscribe(event => console.log(event));

* + Here fromEvent() gets first arg - source of the event, 2nd arg - which event we are subscribing to. And returns Observable<Event>, that is observable which emits Event objects. Unless we call subscribe() method on the Observable, stream will not be created.
  + Subscribe() gets the value(s) that are emitted by the stream itself then an error argument which represents any error when some of the streams might go wrong. So the error argument gives us a way to handle that error. And lastly we can write last argument as a callback function which gets called when the stream is completed without any error.
  + Note that if a stream errors out then we can no longer emit values using that observable.
* According to the observable contract,
  + The observable will emit its values and then it's going to either error out or complete.
  + The observable will either complete or error out but not both.
  + Once an observable completes or errors out, it cannot continue to emit any further values.
  + An observable might continue emitting values without completing or earring out. E.g. interval().
* Important notions about observables is subscriptions and cancellations.
* At one point in time, we may no longer want to handle or get the values that it might be emitting. This is where we need to unsubscribe from that observable (basically calling unsubscribe()).
* We can create our own observables, but make sure that their subscriptions are cancellable, that is when unsubscribe is called on the Subscription object, the subscription should be cancelled.

## Observer

* An observer is the one which emits new values, throws error out to the observable or complete the observable. E.g. observer.next(), observer.error(), observer.complete()
* The observer is what we use internally to implement the observable.
* The observer will only be called when we explicitly subscribe on the Observable.

## RxJS Operator

* Basics about Operator –
  + A pure function which takes one Observable as input and generates another Observable as output.
  + Subscribing to the output Observable will also subscribe to the input Observable.
* Most operators operate on an Observable and return an Observable. This allows you to apply these operators one after the other, in a chain. Each operator in the chain modifies the Observable that results from the operation of the previous operator.
* The order in which the Observable operators appear in the chain does matter.
* The best way to understand any RxJS operator is to check the official documentation and its Mable diagram.
* A chain of Observable operators do not operate independently on the original Observable that originates the chain, but they operate in turn, each one operating on the Observable generated by the operator immediately previous in the chain.
* The way RxJS is meant to be used is we should avoid having a lot of logic in our subscribe calls and we should definitely not nest subscribe calls together. Indeed such imperative implementation is called as RxJS Anti Pattern, which we should avoid. E.g.

courses$.subscribe(

        courses => {

          this.beginnerCourses = courses

                  .filter(course => course.category == 'BEGINNER');

          this.advancedCourses = courses

                  .filter(course => course.category == 'ADVANCED');

        },

        noop,

        () => console.log('completed')

      );

* pipe() function allows us to chain multiple operators.
* of() function allows us to define all sorts of observables.
* concat() function
  + to combine multiple observables and execute them one after another (sequential concatenation)
  + With this, we subscribe to one observable. We emit all its values until it completes and only then we subscribe to further observables in our list that we are trying to concatenate.
  + E.g.

const course1$ = of(1, 2, 3);

    const course2$ = of(4, 5, 6);

    const course3$ = of(7, 8, 9);

    const result$ = concat(course1$, course2$, course3$);

    // subscribe to the concatenated observable

    result$.subscribe(val => console.log(val)); // or just (console.log)

* merge() –
  + merge is ideal for performing synchronous operations in parallel as opposed to sequential in case of concat()
  + Our merge observable is completed when all of the merged observables are completed.
  + On the other hand if any of these observables throw an error then the resulting merged observable is going to err out immediately.
  + Merge strategy is ideal for performing long running operations in parallel and getting the results of each of the operations combined.
  + E.g.

const interval1$ = interval(1000);

    const interval2$ = interval1$.pipe(map(val => val \* 10));

    const merged$ = merge(interval1$, interval2$);

    merged$.subscribe(console.log);

* All RxJS operators are described at –
  + <http://reactivex.io/documentation/operators.html>
  + <https://rxjs-dev.firebaseapp.com/guide/operators>
* map() operator – The Map operator applies a function of your choosing to each item emitted by the source Observable, and returns an Observable that emits the results of these function applications.
* shareReplay() operator –
  + We don’t want our application to do multiple HTTP requests to fetch the exact same data again and again from the backend.
  + The shareReplay() operator solves this problem.
  + As the name implies the operator is going to make sure that our HTTP response is going to be passed on to each new subscription instead of executing the same HTTP request again and again.
* tap() operator –
  + It is meant to be used to produce side effects in our observable chain.
  + So whenever we want to update something outside of the observable chain such as for example updating a variable at the level of the component or to issue a logging statement.
* fromPromise() operator –
  + It takes a promise and creates an RxJS observable derived from that promise.
* concatMap() operator –
  + Take each value of source observable, convert/map it into an observable and on subscribe wait for first observable to complete before initiating next observer.
  + With this, we can control the sequence of events and it is guaranteed that one observable will start only when previous observable is completed.
* mergeMap() operator –
  + Take each value of source observable, convert/map it into an observable and on subscribe start the observables in parallel and outputting the values as they come from each source observable.
  + With this, we cannot control the sequence of events and it is NOT guaranteed that one observable will start only when previous observable is completed.
  + We will only complete the output observable when the source observable has completed.
* exhaustMap() operator –
  + E.g. We can obtain the functionality of ignoring multiple clicks made in the Save button while a save is already ongoing – by using the exhaust map operator instead of concatMap.
  + So the new values are ignored as long as the ongoing observable is not yet completed. Otherwise it is similar to concatMap() operator.
* debounceTime() operator –
  + Emits a value from the source Observable only after a particular time span has passed without another source emission.
  + E.g. It helps to reduce the number of search requests that we are sending to the backend.
  + debounceTime is a really great way for waiting for a stream of values to stabilize before using a given value.
  + Debouncing is about waiting for a value to become stable. If the value doesn’t stabilize for the specified time, then event will not be emitted.
* distinctUntilChanged() operator –
  + Returns an Observable that emits all items emitted by the source Observable that are distinct by comparison from the previous item.
  + So if two consecutive values are exactly the same, we only want to emit one value.
* switchMap() operator –
  + Here we take an input value, create/map to a new observable and on subscribing we emit the values received from the created mapped observable. But if a new value gets emitted before the previous Observable got a chance to complete, the previous inner Observable will be unsubscribed from, and its values will no longer be reflected in the output.
  + switch map is all about the subscription logic.
  + switchMap operator is very commonly used for implementing search type ahead.
  + If the user typed in a new search request while the current search is still ongoing we would prefer that the current search gets immediately cancelled and then we submit a new search. The main logic that we are looking for here is cancellation. We want to cancel the ongoing HTTP request immediately and then switch to a new HTTP request.
* catchError() operator –
  + To catch and handle any error occurred in any of observables in the chain.
  + Catches errors on the observable to be handled by returning a new observable or throwing an error using function throwError().
* throwError() function –
  + Creates an Observable that emits no items to the Observer and immediately emits an error notification.
* finalize() operator –
  + Returns an Observable that mirrors the source Observable, but will call a specified function when the source terminates on complete or error.
  + This can be used to do the cleanup tasks.
* retryWhen() function –
  + To implement retry logic in case of an error.
  + Based on the logic defined in this callback function (retry immediately or retry after some delay), re-subscribes to the source observable.
* delayWhen() function –
  + Delays the emission of items from the source Observable by a given time span determined by the emissions of another Observable.
* startWith() function –
  + The goal of the function is to initialize a stream with a given initial value.
* throttle() operator –
  + throttling is somewhat similar to the debouncing in the sense that we are also trying to reduce the number of values in our stream. But the way that we're doing it is very different.
  + To limit the output rate.
  + Throttle is used for limiting the output by limiting the number of values that can be emitted in a certain interval.
* throttleTime() operator –
  + same as throttle() operator but with handy interval.
* forkJoin() operator –
  + the fork operator allows us to launch several tasks in parallel, wait for those tasks to complete and then we can get back the results of each task and use those combined results together.
* first() operator –
  + gets a stream of values and emits only the first value from the streams and completes.
* take() operator –
  + e.g. take(2)
  + gets a steam of values and emits number of values as defined in the function argument and then the observable is completed.
* withLatestFrom() operator –
  + This operator is especially useful when dealing with long running observables.
  + This operator allows to combine multiple observables together (long running or not) by taking the latest value emitted from each observable and providing that to the next operator in the chain or to the subscribe method as a tuple value.
  + Please note that – withLatestFrom() is all about combining the current value of an observer with the latest value emitted by one or more other observables.

## RxJS Custom Operator

* Basically we need to write a higher order function, that is a function returning another function.

import { Observable } from "rxjs";

import { tap } from "rxjs/operators";

export enum RxJSLoggingLevel {

  TRACE,

  DEBUG,

  INFO,

  ERROR

}

let rxjsApplicationLoggingLevel = RxJSLoggingLevel.INFO;

export function setRxJSApplicationLoggingLevel(level: RxJSLoggingLevel) {

  rxjsApplicationLoggingLevel = level;

}

/\*\*

 \* Here debug is a higher order function - that is a function returning function

 \*/

export const debug = (level: number, message: string) =>

      (source: Observable<any>) => source

          .pipe(

            tap(value => {

              if (level >= rxjsApplicationLoggingLevel) {

                console.log(message + ": ", value);

              }

            })

          );

## Subjects

* It's better to as much as possible create our observables using for example Observable.create() or by using some of the many methods available such as for example from Promise and other methods like of() that allow us to create an observable directly from a source. However if some of those methods are not convenient or if we run into a source of data that is not easily transformable into an observable or if we are doing multicasting of one value to multiple separate observable consumers then we might want to look into the notion of subject.
* A Subject is at the same time an observer and an observable. It has methods from both observer (next(), complete(), error()) and observable (pipe(), subscribe()).
* The subject is meant to be private to the part of the application that is emitting a given set of data.
* We can get the Observable from the Subject using subject.asObservable() and we can then share this Observable in other parts of the application (unlike the Subject itself because we don’t want other parts of our application to emit the events)
* A subject is a very convenient way and much easier to understand of creating a custom observable. There are a couple of differences though.
* We don't have any way of providing unsubscribe logic to our observable that gets derived from the subject. And we also run the risk of sharing accidentally the subject with other parts of the application which means that those other parts of the application could potentially take over the behavior of the observable by directly calling the next(), complete() or error() of the subject which is not intended.
* By this reason, we should try to use subjects as little as possible. Instead we should try to derive our observables directly from the source as much as possible using methods such as for example fromPromise() to derive an observable from a promise or simply from from() method that allows us to derive an observable directly from a browser event for example.
* However if by some reason that is not practical or even possible then using a subject is a great way of creating a custom observable.
* Another very common use case for our just subjects is multicasting. In the case of multicasting we want to take one value from one observable stream and remit that into multiple separate output streams.
* Subjects maintain a registry of many listeners/subscriber.
* Unicast:
  + A transmission/stream sends IP packets to a single recipient on a network.
  + Example : Plain Observable
  + Explanation: you have to do .subscribe(). If you subscribe multiple times against the original observable, you are not sharing the original observable. You are creating a new observable every time you subscribe.
* Multicast:
  + A transmission sends IP packets to a group of hosts on a network.
  + Example: RxJS Subject, Promises
  + Explanation: You have to do .subscribe() as many times as you want against that original observable.

### Types of Subjects

#### Plain Subjects

* The default behavior of plain subject is – On subscription to the subject, we will receive new values that are made after the subscription. But we will not get access the previously emitted values.
* E.g. const subject = new Subject();
* The problem is that when we are writing a synchronous problems we very often want our late subscribers to receive something from the observable. Typically we want our late subscribers to receive the latest value emitted by the observable.

#### Behavioral Subjects

* Similar to the plain subject but it also supports late subscriptions.
* The goal of behavioral subject is to always provide a value to subscribers even if the subscription happens late.
* Since the goal is to always provide a value to subscribers we also need to pass in here an initial value.
* E.g. const subject = new BehaviorSubject(0);
* So for early subscribers (who subscribe before any event/value is emitted by the Subject) will get the default value defined and of course subsequent emitted values and in case of the late subscribers (who subscribe at later some point after one or more events/values are emitted) will receive the last emitted value (and of course subsequent emitted values) as long as the subject is not completed or erred out.
* Also if the completion happens before a subscription takes place then that late subscriber will not receive any value.
* Behavioral subject is probably the most commonly used type of subject.

#### Async Subjects

* Async subject will wait for observable completion before emitting any of the values to the multiple subscribers. The value emitted is going to be the last value on completion. If that subject is not completed, then no value will be emitted.
* Both early and late subscribers will receive the last emitted value on completion.
* Async subject is ideal for using with long running calculations where we have an observable that is emitting a lot of intermediate calculation values. Let's say that the calculation is ongoing and we are progressively reporting the latest most up to date value of that calculation but the calculation has not finished yet. When the calculation is finished then the last value of the subject is going to be emitted and then the subject is going to be completed. In such scenarios, the subscriber is not interested in intermediate values but on the final value on completion. So in such cases use Async Subjects.
* Async subject is ideal for handling long running calculations where we really want only to receive the last value.

#### Replay Subjects

* Sometimes a late subscriber might also be interested to receive not just the last emitted value but also all the intermediate values. In such cases, we can use Replay Subjects.
* So the logic here is not linked to the observer completion. Even if the observer is completed, the last subscriber will receive all the intermediate emitted values.

#### Example to demonstrate all these subjects

ngOnInit() {

  const subject = new ReplaySubject(); // Subject(), BehaviorSubject(0), AsyncSubject()

  const series$ = subject.asObservable();

  series$.subscribe(val => console.log('first '+val));

  subject.next(1);

  subject.next(2);

  subject.next(3);

  subject.complete();

  setTimeout(() => {

    series$.subscribe(val => console.log('late '+val));

  }, 2000);

}

## Store

* Each time we navigate to a url/route, we always try to fetch data from backend using the HTTP calls. This way most of the times we unnecessarily give duplicate HTTP calls to the backend. Instead, we could keep the once retrieved data into a central store at client side. This is what the store design pattern is.
* So whenever a component gets discarded our data should not get discarded with it. We need a central place in memory on the client to store our data. And whenever our component needs the data, it simply needs to subscribe to it and it's going to receive the latest version of the data.
* So our Store design will have a centralized service that is going to contain our data and that service is going to expose a couple of observables that service is going to be responsible for fetching the data from the back and at the appropriate moment. And also it's going to be responsible for storing that data in memory providing it to the rest of the application under the form of an observable.