**[RxJS](https://github.com/ReactiveX/rxjs" \t "_blank)** is one of the hottest libraries in web development today. Offering a powerful, functional approach for dealing with events and with integration points into a growing number of frameworks, libraries, and utilities, the case for learning Rx has never been more appealing. Couple this with the ability to utilize your knowledge across [nearly any language](http://reactivex.io/languages.html), having a solid grasp on reactive programming and what it can offer seems like a no-brainer.

Reactive Extensions for JavaScript is a library for transforming, composing, and querying streams of data. We mean all kinds of data too, from simple arrays of values, to series of events (unfortunate or otherwise), to complex flows of data.

**Stream**: A stream in the RxJS world simply represents **values over time**. Users sending chat messages, a user clicking around on a page, a user filling out different form fields in a form;

The essential concepts in RxJS which solve async event management are:

* Observable
* Observer
* Subscription
* Operators
* Subject
* Schedulers

**Observables**

**Observables** help you manage asynchronous data, such as data coming from a backend service. ... To use **observables**, Angular uses a third-party library called Reactive Extensions (**RxJS**). **Observables** are a proposed feature for ES 2016

Observables are a Stream, Sequence of data over time.

We can create an observable in 13 ways using operators that we look in later sections.

Let’s create it using one of them

var foo = Rx.Observable.create(function (observer) {

console.log('Hello');

observer.next(42);

});

foo.subscribe(function (x) {

console.log(x);

});

foo.subscribe(function (y) {

console.log(y);

});

And the output is the same:

"Hello"

42

"Hello"

42

Observables will get executed independently for each subscription.

## Anatomy of an Observable

Observables are **created** using Rx.Observable.create or a creation operator, are **subscribed** to with an Observer, **execute** to deliver next / error / complete notifications to the Observer, and their execution may be **disposed**. These four aspects are all encoded in an Observable instance, but some of these aspects are related to other types, like Observer and Subscription.

Core Observable concerns:

* **Creating** Observables
* **Subscribing** to Observables
* **Executing** the Observable
* **Disposing** Observables

### Creating Observables

Rx.Observable.create is an alias for the Observable constructor, and it takes one argument: the subscribe function.

The following example creates an Observable to emit the string 'hi' every second to an Observer.

var observable = Rx.Observable.create(function subscribe(observer) {

var id = setInterval(() => {

observer.next('hi')

}, 1000);

});

### Subscribing to Observables

The Observable observable in the example can be subscribed to, like this:

observable.subscribe(x => console.log(x));

### Executing Observables

There are three types of values an Observable Execution can deliver:

* "Next" notification: sends a value such as a Number, a String, an Object, etc.
* "Error" notification: sends a JavaScript Error or exception.
* "Complete" notification: does not send a value.

var observable = Rx.Observable.create(function subscribe(observer) {

try {

observer.next(1);

observer.next(2);

observer.next(3);

observer.complete();

} catch (err) {

observer.error(err); // delivers an error if it caught one

}

});

### Disposing Observable Executions

var observable = Rx.Observable.from([10, 20, 30]);

var subscription = observable.subscribe(x => console.log(x));

// Later:

subscription.unsubscribe();

**Hot vs Cold**

**Cold observables** start running on subscription. Values are not shares among the subscribers. By default observables are cold

**Hot observables**

Const source = Rx.Observable.interval(1000).publish();

Source.connect();

Source.subscribe(values => console.log(value));

Source.subscribe(values => console.log(value));

Observables are cold, or unicast by default. Multicast operators can used to make observable hot

**Publish**: Return a ConnectableObservable, which is a variety of observable waits until its connect method is called before it emits items to observer tha have subscribed to it.

**Subscribing** to changes (Observer)

To listen to data on an observable, we must call the subscribe()method. This method returns a subscription, which we can use later tostop reacting to the incoming data if we are no longer interested in it.

The subscribe()method of observables has the following signature:

observable.subscribe(onNext,onError,onCompleted);

All parameters are optional and can be omitted if we are not interested in this type of even

onNext: This is a function to be called every time new data is propagated through the observable

onError: This is a function to be called every time an error occurs in the observable

onCompleted: This is a function to be called when the observable is completed

#### Operators

Operators are the horse-power behind observables, providing an elegant, declarative solution to complex asynchronous tasks