RX JS with React and Redux

By Murthy

Reactive Extensions also can be used in React and Redux to work with asynchronous stream data handling.

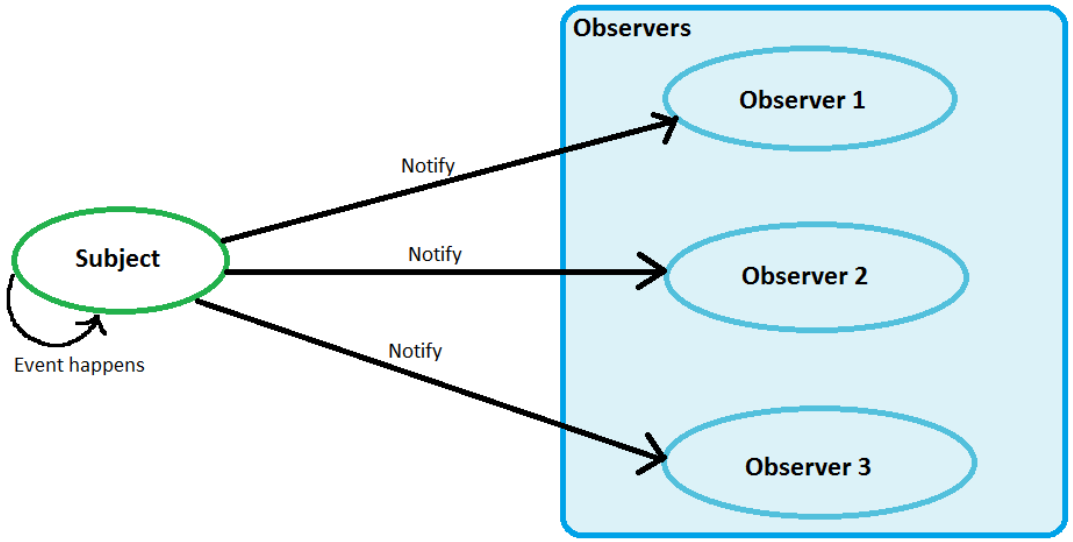
Redux-Observable is an RxJS-based middleware for Redux that allows developers to work with async actions. It's an alternative to redux-thunk and redux-saga.

Observer Pattern

In Observer pattern, an object called "Observable" or "Subject", maintains a collection of subscribers called "Observers." When the subjects' state changes, it notifies all its Observers.

In JavaScript, the simplest example would be event emitters and event handlers.

When we do .addEventListener, we are pushing an observer into the subject's collection of observers. Whenever the event happens, the subject notifies all the observers.



Observer Pattern

RxJS

RxJS is JavaScript implementation of ReactiveX, a library for composing asynchronous and event-based programs by using observable sequences.

RxJS is an implementation of the Observer pattern. It also extends the Observer pattern by providing operators that allow us to compose Observables and Subjects in a declarative manner.

Observers, Observables, Operators, and Subjects are the building blocks of RxJS.

Observers

Observers are objects that can subscribe to Observables and Subjects. After subscribing, they can receive notifications of three types - next, error, and complete.

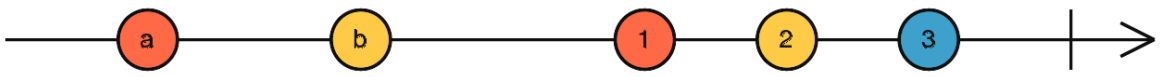
Any object with the following structure can be used as an Observer.

```
interface Observer<T> {
  closed?: boolean;
  next: (value: T) => void;
  error: (err: any) => void;
  complete: () => void;
}
```

When the Observable pushes next, error, and complete notifications, the Observer's .next, .error, and .complete methods are invoked.

Observables

Observables are objects that can emit data over a period of time. It can be represented using the "marble diagram".



Successfully completed Observable

Where the horizontal line represents the time, the circular nodes represent the data emitted by the Observable, and the vertical line indicates that the Observable has completed successfully.

Observable with an error

Observables may encounter an error. The cross represents the error emitted by the Observable.

The "completed" and "error" states are final. That means, Observables cannot emit any data after completing successfully or encountering an error.

Creating an Observable

Observables are created using the new Observable constructor that takes one argument - the subscribe function. Observables can also be created using some operators, but we will talk about that later when we talk about Operators.

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

const observable = new Observable(subscriber => {
  // Subscribe function
});
```

Subscribing to an Observable

Observables can be subscribed using their .subscribe method and passing an Observer.

```
observable.subscribe({
  next: (x) => console.log(x),
  error: (x) => console.log(x),
  complete: () => console.log('completed');
});
```

Execution of an Observable

The subscribe function that we passed to the new Observable constructor is executed every time the Observable is subscribed.

The subscribe function takes one argument - the Subscriber. The Subscriber resembles the structure of an Observer, and it has the same 3 methods: .next, .error, and .complete.

Observables can push data to the Observer using the .next method. If the Observable has completed successfully, it can notify the Observer using the .complete method. If the Observable has encountered an error, it can push the error to the Observer using the .error method.

```
// Create an Observable
const observable = new Observable(subscriber => {
  subscriber.next('first data');
  subscriber.next('second data');
  setTimeout(() => {
    subscriber.next('after 1 second - last data');
    subscriber.complete();
    subscriber.next('data after completion'); // <-- ignored
  }, 1000);
  subscriber.next('third data');
});

// Subscribe to the Observable
observable.subscribe({
   next: (x) => console.log(x),
   error: (x) => console.log(x),
   complete: () => console.log('completed')
});

// Outputs:
//
// first data
// second data
// third data
// after 1 second - last data
// completed
```

Observables are Unicast

Observables are *unicast*, which means Observables can have at most one subscriber. When an Observer subscribes to an Observable, it gets a copy of the Observable that has its own execution path, making the Observables unicast.

Example: let us create an Observable that emits 1 to 10 over 10 seconds. Then, subscribe to the Observable once immediately, and again after 5 seconds.

```
// Create an Observable that emits data every second for 10 seconds
const observable = new Observable(subscriber => {
        let count = 1;
    const interval = setInterval(() => {
                subscriber.next(count++);

        if (count > 10) {
            clearInterval(interval);
        }
    }, 1000);
});

// Subscribe to the Observable
observable.subscribe({
        next: value => {
        console.log(`Observer 1: ${value}`);
    }
});

// After 5 seconds subscribe again
setTimeout(() => {
    observable.subscribe({
        next: value => {
            console.log(`Observer 2: ${value}`);
        }
    });
}, 5000);

/* Output

Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 1: 4
Observer 1: 5
Observer 2: 1
```

```
Observer 1: 6
Observer 2: 2
Observer 1: 7
Observer 2: 3
Observer 1: 8
Observer 2: 4
Observer 1: 9
Observer 2: 5
Observer 1: 10
Observer 2: 6
Observer 2: 7
Observer 2: 8
Observer 2: 9
Observer 2: 10

*/
```

In the output, you can notice that the second Observer started printing from 1 even though it subscribed after 5 seconds. This happens because the second Observer received a copy of the Observable whose subscribe function was invoked again. This illustrates the unicast behaviour of Observables.

Subjects

A Subject is a special type of Observable.

Creating a Subject

A Subject is created using the new Subject constructor.

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

// Create a subject
const subject = new Subject();
```

Subscribing to a Subject

Subscribing to a Subject is similar to subscribing to an Observable: you use the .subscribe method and pass an Observer.

```
subject.subscribe({
  next: (x) => console.log(x),
  error: (x) => console.log(x),
  complete: () => console.log("done")
});
```

Execution of a Subject

Unlike Observables, a Subject calls its own .next, .error, and .complete methods to push data to Observers.

```
// Push data to all observers
subject.next('first data');

// Push error to all observers
subject.error('oops something went wrong');

// Complete
subject.complete('done');
```

Subjects are Multicast

Subjects are *multicast:* multiple Observers share the same Subject and its execution path. It means all notifications are broadcasted to all the Observers. It is like watching a live program. All viewers are watching the same segment of the same content at the same time.

Example: let us create a Subject that emits 1 to 10 over 10 seconds. Then, subscribe to the Observable once immediately, and again after 5 seconds.

```
// Create a subject
const subject = new Subject();

let count = 1;
const interval = setInterval(() => {
   subscriber.next(count++);
   if (count > 10) {
      clearInterval(interval);
   }
}, 1000);

// Subscribe to the subjects
subject.subscribe(data => {
   console.log(`Observer 1: ${data}`);
});

// After 5 seconds subscribe again
```

```
setTimeout(() => {
   subject.subscribe(data => {
          console.log(`Observer 2: ${data}`);
          });
}, 5000);

/* OUTPUT

Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 1: 4
Observer 1: 5
Observer 2: 5
Observer 1: 6
Observer 2: 6
Observer 1: 7
Observer 2: 7
Observer 1: 8
Observer 2: 8
Observer 1: 9
Observer 2: 9
Observer 1: 10
Observer 2: 10

*/
```

In the output, notice that the second Observer started printing from 5 instead of starting from 1. This happens because the second Observer is sharing the same Subject. Since it subscribed after 5 seconds, the Subject has already finished emitting 1 to 4. This illustrates the multicast behavior of a Subject.

Subjects are both Observable and Observer

Subjects have the .next, .error and .complete methods. That means that they follow the structure of Observers. Hence, a Subject can also be used as an Observer and passed to the .subscribe function of Observables or other Subjects.

Example: let us create an Observable and a Subject. Then subscribe to the Observable using the Subject as an Observer. Finally, subscribe to the Subject. All the values emitted by the Observable will be pushed to the Subject, and the Subject will broadcast the received values to all its Observers.

```
// Create an Observable that emits data every second
const observable = new Observable(subscriber => {
  let count = 1;
  const interval = setInterval(() => {
     subscriber.next(count++);

     if (count > 5) {
          clearInterval(interval);
     }
  }, 1000);
});

// Create a subject
const subject = new Subject();

// Use the Subject as Observer and subscribe to the Observable
observable.subscribe(subject);

// Subscribe to the subject
subject.subscribe({
   next: value => console.log(value)
});

/* Output

1
2
3
4
5

*/
```

Operators

Operators are what make RxJS useful. Operators are pure functions that return a new Observable. They can be categorized into 2 main categories:

1. Creation Operators
2. Pipeable Operators

Creation Operators

Creation Operators are functions that can create a new Observable.

Example: we can create an Observable that emits each element of an array using the from operator.

```
const observable = from([2, 30, 5, 22, 60, 1]);

observable.subscribe({
   next: (value) => console.log("Received", value),
   error: (err) => console.log(err),
   complete: () => console.log("done")
});

/* OUTPUTS

Received 2
Received 30
Received 5
Received 22
Received 60
Received 1
done

*/
```

The same can be an Observable using the marble diagram.

Pipeable Operators

Pipeable Operators are functions that take an Observable as an input and return a new Observable with modified behavior.

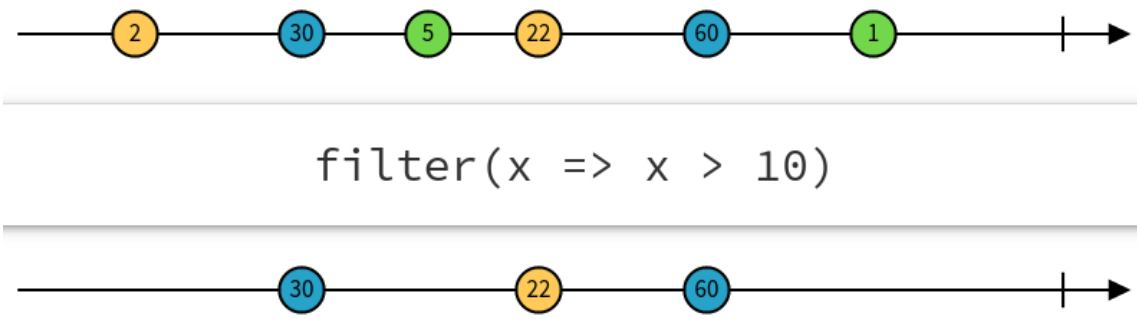
Example: let's take the Observable that we created using the from operator. Now using this Observable, we can to create a new Observable that emits only numbers greater than 10 using the filter operator.

```
const greaterThanTen = observable.pipe(filter(x => x > 10));

greaterThanTen.subscribe(console.log, console.log, () => console.log("completed"));

// OUTPUT
// 11
// 12
// 13
// 14
// 15
```

The same can be represented using the marble diagram.



There are many more useful operators out there. You can see the full operators list along with examples on the official RxJS documentation here.
It is crucial to understand all the commonly used operators. Here are some operators that I use often:

1. mergeMap
2. switchMap
3. exhaustMap
4. map
5. catchError
6. startWith
7. delay
8. debounce

9. throttle
10. interval
11. from
12. of

Redux Observables
As per the official website,

RxJS-based middleware for Redux. Compose and cancel async actions to create side effects and more.

In Redux, whenever an action is dispatched, it runs through all the reducer functions and a new state is returned.

Redux-observable takes all these dispatched actions and new states and creates two observables out of it - Actions observable action$, and States observable state$.
Actions observable will emit all the actions that are dispatched using the store.dispatch(). States observable will emit all the new state objects returned by the root reducer.

Epics
It is a function which takes a stream of actions and returns a stream of actions. Actions in, actions out.

Epics are functions that can be used to subscribe to Actions and States Observables. Once subscribed, epics will receive the stream of actions and states as input, and it must return a stream of actions as an output. *Actions In - Actions Out.*

```
const someEpic = (action$, state$) => {
  return action$.pipe( // subscribe to actions observable
    map(action => { // Receive every action, Actions In
      return someOtherAction(); // return an action, Actions Out
    })
  )
}
```

It is important to understand that all the actions received in the Epic have already *finished running through the reducers*.
Inside an Epic, we can use any RxJS observable patterns, and this is what makes redux-observables useful.

Example: we can use the .filter operator to create a new intermediate observable. Similarly, we can create any number of intermediate observables, but the final

output of the final observable must be an action, otherwise an exception will be raised by redux-observable.

```
const sampleEpic = (action$, state$) => {
   return action$.pipe(
          filter(action => action.payload.age >= 18), // can create intermediate
observables and streams
      map(value => above18(value)) // where above18 is an action creator
   );
}
```

Every action emitted by the Epics are immediately dispatched using the store.dispatch().

Setup

First, let's install the dependencies.

```
npm install --save rxjs redux-observable
```

Create a separate folder named epics to keep all the epics. Create a new file index.js inside the epics folder and combine all the epics using the combineEpics function to create the root epic. Then export the root epic.

```
import { combineEpics } from 'redux-observable';
import { epic1 } from './epic1';
import { epic2 } from './epic2';

const epic1 = (action$, state$) => {
 ...
}

const epic2 = (action$, state$) => {
 ...
}

export default combineEpics(epic1, epic2);
```

Create an epic middleware using the createEpicMiddleware function and pass it to the createStore Redux function.

```
import { createEpicMiddleware } from 'redux-observable';
import { createStore, applyMiddleware } from 'redux';
import rootEpic from './rootEpics';

const epicMiddleware = createEpicMiddlware();
```

```
const store = createStore(
  rootReducer,
  applyMiddleware(epicMiddlware)
);
```

Finally, pass the root epic to epic middleware's .run method.

```
epicMiddleware.run(rootEpic);
```

Some Practical Usecases

RxJS has a big learning curve, and the redux-observable setup worsens the already painful Redux setup process. All that makes Redux observable look like an overkill. But here are some practical use cases that can change your mind.

Throughout this section, I will be comparing redux-observables with redux-thunk to show how redux-observables can be helpful in complex use-cases. I don't hate redux-thunk, I love it, and I use it every day!

1. Make API Calls

Usecase: Make an API call to fetch comments of a post. Show loaders when the API call is in progress and also handle API errors.
A redux-thunk implementation will look like this,

```
function getComments(postId){
  return (dispatch) => {
    dispatch(getCommentsInProgress());
    axios.get(`/v1/api/posts/${postId}/comments`).then(response => {
      dispatch(getCommentsSuccess(response.data.comments));
    }).catch(() => {
      dispatch(getCommentsFailed());
    });
  }
}
```

and this is absolutely correct. But the action creator is bloated.

We can write an Epic to implement the same using redux-observables.

```
const getCommentsEpic = (action$, state$) => action$.pipe(
  ofType('GET_COMMENTS'),
  mergeMap((action) =>
from(axios.get(`/v1/api/posts/${action.payload.postId}/comments`).pipe(
    map(response => getCommentsSuccess(response.data.comments)),
    catchError(() => getCommentsFailed()),
```

```
      startWith(getCommentsInProgress())
    )
);
```

Now it allows us to have a clean and simple action creator like this,

```
function getComments(postId) {
  return {
    type: 'GET_COMMENTS',
    payload: {
      postId
    }
  }
}
```

2. Request Debouncing

Usecase: Provide autocompletion for a text field by calling an API whenever the value of the text field changes. API call should be made 1 second after the user has stopped typing.

A redux-thunk implementation will look like this,

```
let timeout;

function valueChanged(value) {
  return dispatch => {
    dispatch(loadSuggestionsInProgress());
    dispatch({
      type: 'VALUE_CHANGED',
      payload: {
        value
      }
    });

    // If changed again within 1 second, cancel the timeout
    timeout && clearTimeout(timeout);

    // Make API Call after 1 second
    timeout = setTimeout(() => {
        axios.get(`/suggestions?q=${value}`)
         .then(response =>
```

```
                dispatch(loadSuggestionsSuccess(response.data.suggestions)))
            .catch(() => dispatch(loadSuggestionsFailed()))
      }, 1000, value);
   }
}
```

It requires a global variable timeout. When we start using global variables, our action creators are not longer pure functions. It also becomes difficult to unit test the action creators that use a global variable.
We can implement the same with redux-observable using the .debounce operator.

```
const loadSuggestionsEpic = (action$, state$) => action$.pipe(
   ofType('VALUE_CHANGED'),
   debounce(1000),
   mergeMap(action =>
from(axios.get(`/suggestions?q=${action.payload.value}`)).pipe(
         map(response => loadSuggestionsSuccess(response.data.suggestions)),
      catchError(() => loadSuggestionsFailed())
   )),
   startWith(loadSuggestionsInProgress())
);
```

Now, our action creators can be cleaned up, and more importantly, they can be pure functions again.

```
function valueChanged(value) {
   return {
      type: 'VALUE_CHANGED',
      payload: {
         value
      }
   }
}
```

3. Request Cancellation

Usecase: Continuing the previous use-case, assume that the user didn't type anything for 1 second, and we made our 1st API call to fetch the suggestions.
Let's say the API itself takes an average of 2-3 seconds to return the result. Now, if the user types something while the 1st API call is in progress, after 1 second, we will make our 2nd API. We can end up having two API calls at the same time, and it can create a race condition.

To avoid this, we need to cancel the 1st API call before making the 2nd API call.

A redux-thunk implementation will look like this,

```
let timeout;
var cancelToken = axios.cancelToken;
let apiCall;

function valueChanged(value) {
   return dispatch => {
      dispatch(loadSuggestionsInProgress());
      dispatch({
         type: 'VALUE_CHANGED',
         payload: {
            value
         }
      });

      // If changed again within 1 second, cancel the timeout
      timeout && clearTimeout(timeout);

      // Make API Call after 1 second
      timeout = setTimeout(() => {
         // Cancel the existing API
         apiCall && apiCall.cancel('Operation cancelled');

         // Generate a new token
         apiCall = cancelToken.source();


         axios.get(`/suggestions?q=${value}`, {
            cancelToken: apiCall.token
         })
            .then(response =>
dispatch(loadSuggestionsSuccess(response.data.suggestions)))
            .catch(() => dispatch(loadSuggestionsFailed()))

      }, 1000, value);
   }
}
```

Now, it requires another global variable to store the Axios's cancel token. More global variables = more impure functions!

To implement the same using redux-observable, all we need to do is replace .mergeMap with .switchMap.

```
const loadSuggestionsEpic = (action$, state$) => action$.pipe(
  ofType('VALUE_CHANGED'),
  throttle(1000),
  switchMap(action =>
from(axios.get(`/suggestions?q=${action.payload.value}`)).pipe(
        map(response => loadSuggestionsSuccess(response.data.suggestions)),
    catchError(() => loadSuggestionsFailed())
  )),
  startWith(loadSuggestionsInProgress())
);
```

Since it doesn't require any changes to our action creators, they can continue to be pure functions.