RxJS in Practice

Writing our own Ngrx

Reactive Fox Follow
Nov 18, 2018 · 9 min read

> ***Redux*** *is not the only* ***state manager****, and in fact, we can easily create our own. All popular state managers for* ***Angular*** *require to put your business logic inside a singleton. Therefore, I urge you to think carefully when it comes to choosing what state management solution to use.*

State foxement

I'm writing this post because I'm seeing an improper usage of **RxJS** in people's hands everywhere. The most common issues here are not knowing operators, Rx design principles or lack of understanding of declaratively and reactively. In this post, we will cover the most common cases through writing our own **Ngrx** using **RxJS**.

What we want to accomplish

— preserving current state;
— changing the state;
— handling of various actions;
— async stuff;
— error processing;
and last but not least, destroying the state when we **no longer need it**.

. . .

A State of the State

For the sake of the example, we will store a simple list of numbers and loading indicator. Here's the interface:

```
interface ItemsState {
  items: number[];
  loading: boolean;
}
```

Let's define default state:

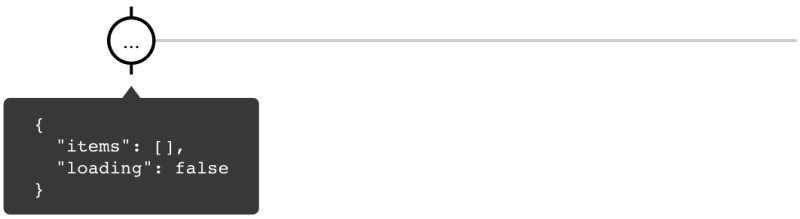
```
const defaultState: ItemsState = {
  items: [],
  loading: false
};
```

of()

In order to be able to work with our state, we can use **of()** operator to create an **Observable**.

of() creates a stream with one or more than one element which completes right after all elements are sent.

```
state$: Observable<ItemsState> = of(defaultState);
```



https://rxviz.com/v/XJzKNLX8

As we can see from the diagram, **Observable** returns our default state and completes. Let's make the stream infinite.

NEVER, startWith()

To keep the stream alive we can use **Subject**, but first, let's take a look at **NEVER** constant. We will touch **Subject** later, no worries.

NEVER is a simple stream in RxJS that never completes.

startWith() creates initial value for the stream. Combined with **NEVER** it can replace **of()** operator.

```
state$: Observable<ItemsState> =
  NEVER.pipe(
    startWith(defaultState)
  );
```

https://rxviz.com/v/xOvKQRpJ

Note, now our stream never ends but every subscriber will work with **different streams** which means that they also will have **different data**. Next, we're going to solve this problem.

publishReplay(), refCount()

BehaviorSubject is usually used when we need to have a state stream. In our example, the best way to go will be using **publishReplay()** and **refCount()**.

publishReplay() creates a message buffer and takes the size of the buffer as its first argument. New subscribers will instantly get those buffered messages. In our case, we need to store only the last message, so we will pass 1.

refCount() implements a simple Ref Count pattern which is used to determine if the stream is alive, meaning that it has subscribers, or not. If there are no subscribers, **refCount()** will unsubscribe from it, thus killing the stream.

```
state$: Observable<ItemsState> =
  NEVER.pipe(
    startWith(defaultState),
    publishReplay(1),
    refCount()
  );
```

https://rxviz.com/v/58GYqgvO

This way we can ensure that all subscribers have the same stream and the same data.

. . .

Control stream that changes the State

Let's define how we want to control it. One way of controlling the state is creating and processing commands. The interface looks like that:

```
interface Action {
  type: string,
  payload?: any
}
```

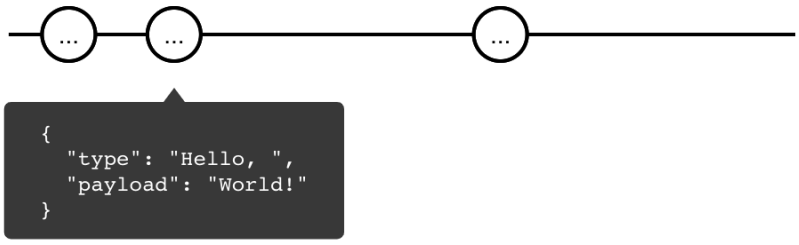
Type property contains command name, payload carries necessary data for the command.

Subject

Now we're going to implement command stream, and **Subject**, mentioned above, is the perfect candidate here. It will create a bidirectional stream that not only can be readable but also writable.

We will create the stream of commands called **actions$** using **Subject**.

```
actions$: Subject<Action> = new Subject<Action>();
```



https://rxviz.com/v/qJyAK9aJ

We created commands stream here, let's bind it with the state stream by replacing **NEVER** with **actions$**.

```
actions$: Subject<Action> = new Subject<Action>();

state$: Observable<ItemsState> =
  actions$.pipe(
    startWith(defaultState),
    publishReplay(1),
```

```
      refCount()
    );
```

https://rxviz.com/v/QJVYLPNO

Now we have two streams: state stream and command stream. They interact with each other but our state is just getting rewritten on every command.

. . .

Command handling

To handle the command we should get the state and command from a stream, change the state and return a new one. We have **scan()** operator to deal with such things.

scan()

scan() receives a reducer function that takes current state and new command from the stream.

Here we're implementing reducer function and passing it to **scan()**.

```
function stateReducer(
  state: ItemsState,
  action: Action
): ItemsState => {
  switch (action.type) {
    default:
      return state;
  }
}
```

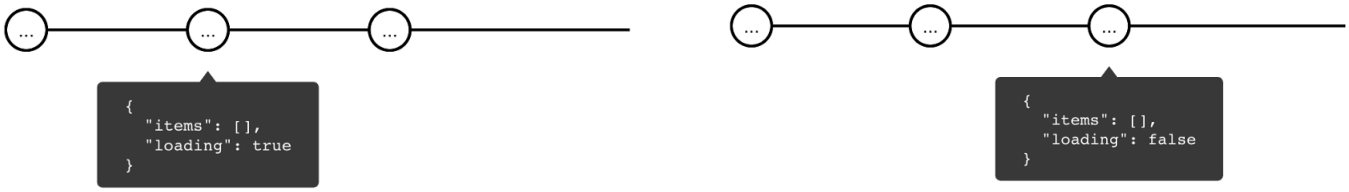
```
state$: Observable<ItemsState> =
  actions$.pipe(
    startWith(defaultState),
    scan(stateReducer),
    publishReplay(1),
    refCount()
  );
```

https://rxviz.com/v/XJzKNM68

Now, the stream is holding its state but does not react to changes. Here's how we add handling for **load** and **load success**:

```
function stateReducer(
  state: ItemsState,
  action: Action
): ItemsState => {
  switch (action.type) {
    case 'load':
      return { ...state, loading: true };
    case 'load success':
      return { ...state, loading: false };
    default:
      return state;
```

```
    }
  }

  state$: Observable<ItemsState> =
    actions$.pipe(
      startWith(defaultState),
      scan(stateReducer),
      publishReplay(1),
      refCount()
    );
```

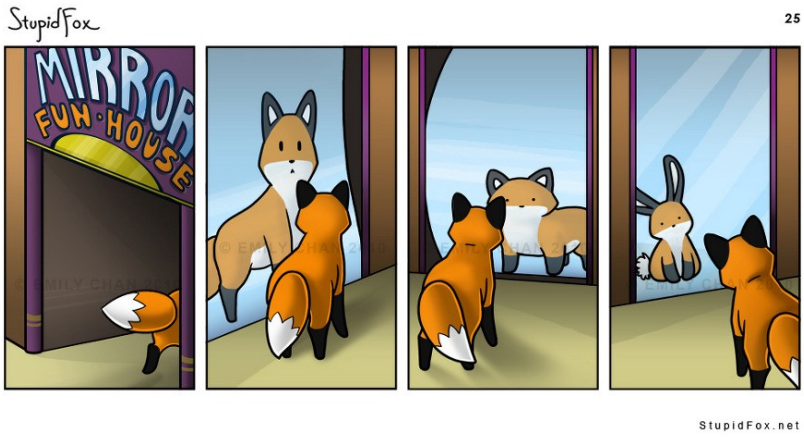


https://rxviz.com/v/38jdvAYO

The state changes to **loading: true** or **loading: false** on **load** and **load success** commands, respectively.

. . .

Effect handling



Our state can react to **synchronous** commands. What should we do with **asynchronous** ones? We need a stream which will take the

command and return new command. Here it is:

```
load$: Observable<Action> = actions$;
```

filter()

First, we need to ensure that initial command has load type. We will use **filter()** operator for that.

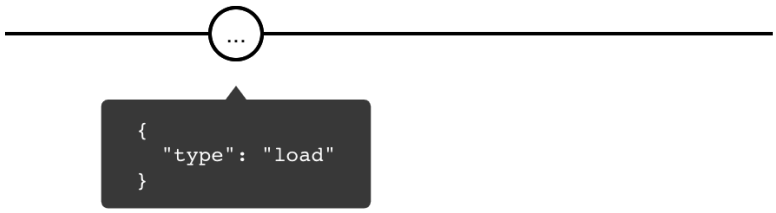
filter() decides whether the command can be passed down the stream or not.

```
load$: Observable<Action> =
  actions$.pipe(
    filter((action) => 'load' === action.type)
  );
```

To make the code more readable, we will create a custom **RxJS** operator. It's considered a good practice. We need an operator that will take a type of command and filter out others.

```
function ofType<T extends Action>(
  type: string
): MonoTypeOperatorFunction<T> {
  return filter((action) => type === action.type);
}
```

```
load$: Observable<Action> =
  actions$.pipe(
    ofType('load')
  );
```



https://rxviz.com/v/moY1ZEKo

Now we have a separate stream that receives commands of a particular type, and we're going to use it to load data asynchronously. For the sake of simplicity, we will emulate loading over the network using a predefined value and **delay()**.

delay()

As the name implies, **delay()** suspends execution of the operators' chain for a specified time, we're using 1 second here.

```
function load(): Observable<number[]> {
  return of([ 1, 2, 3 ]).pipe(
    delay(1000)
  );
}
```

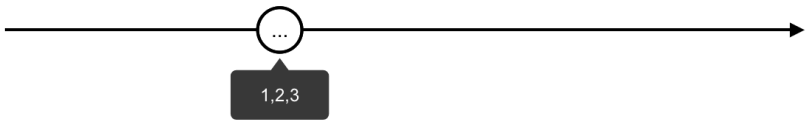
https://rxviz.com/v/58GYqA4O

Now let's take out **load()** function and put it inside **switchMap()**.

switchMap()

switchMap() creates a stream each time it receives a value. If at the moment of receiving a new message, it's already working on the message, it ends the old stream.

```
load$: Observable<Action> =
  actions$.pipe(
    ofType('load'),
    switchMap(() => load())
  );
```



https://rxviz.com/v/7JXXaK6J

Currently, **load$** stream returns data from **load()** function, and so we can finally create **load success** command with our data residing in payload property. We will use **map()** to achieve that.

map()

map() takes data from a stream, changes it and then returns back changed to the stream.

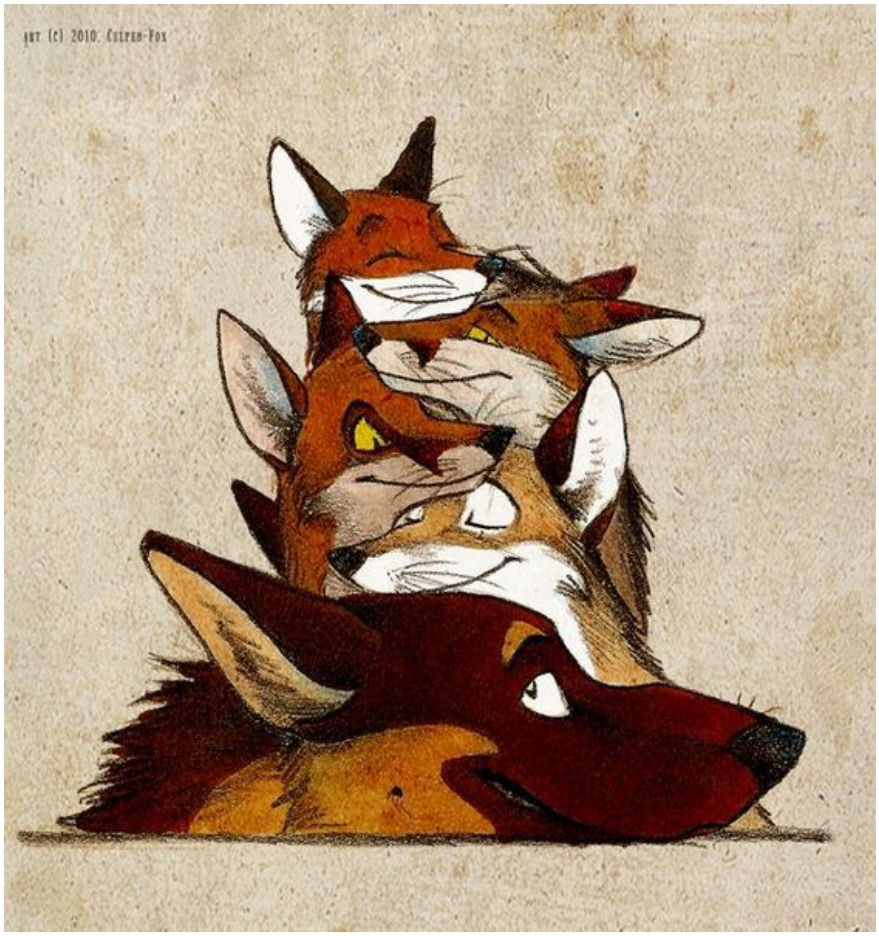
```
load$: Observable<Action> =
  actions$.pipe(
    ofType('load'),
    switchMap(() => load()),
    map((data): Action => ({
      type: 'load success',
      payload: data
    }))
  );
```

https://rxviz.com/v/RoQ7y2qJ

So, we have an effect that receives command, loads data and returns it in the right form.

. . .

Getting everything together

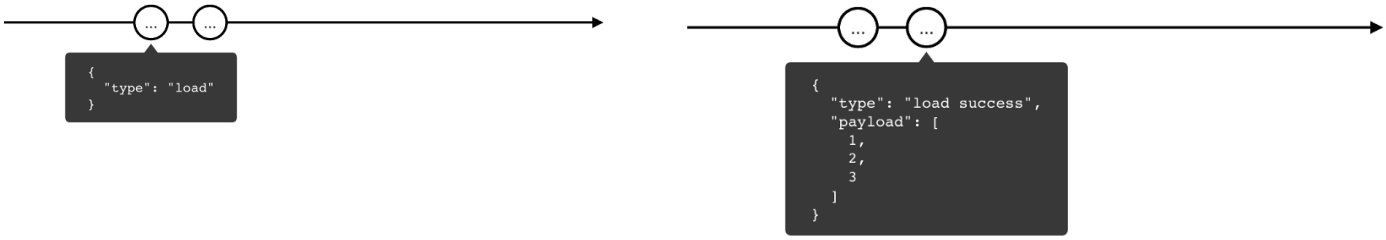


Before we will move to the implementation of **load success** command, we need to make some changes. We should remove direct dependency between **state$** and **actions$**. It can be done by creating new **dispatcher$** stream that just merges all messages from **state$** and **load$**. Here comes the last operator in this post: **merge()**.

merge()

merge() takes messages from all streams and puts them into one stream which it returns.

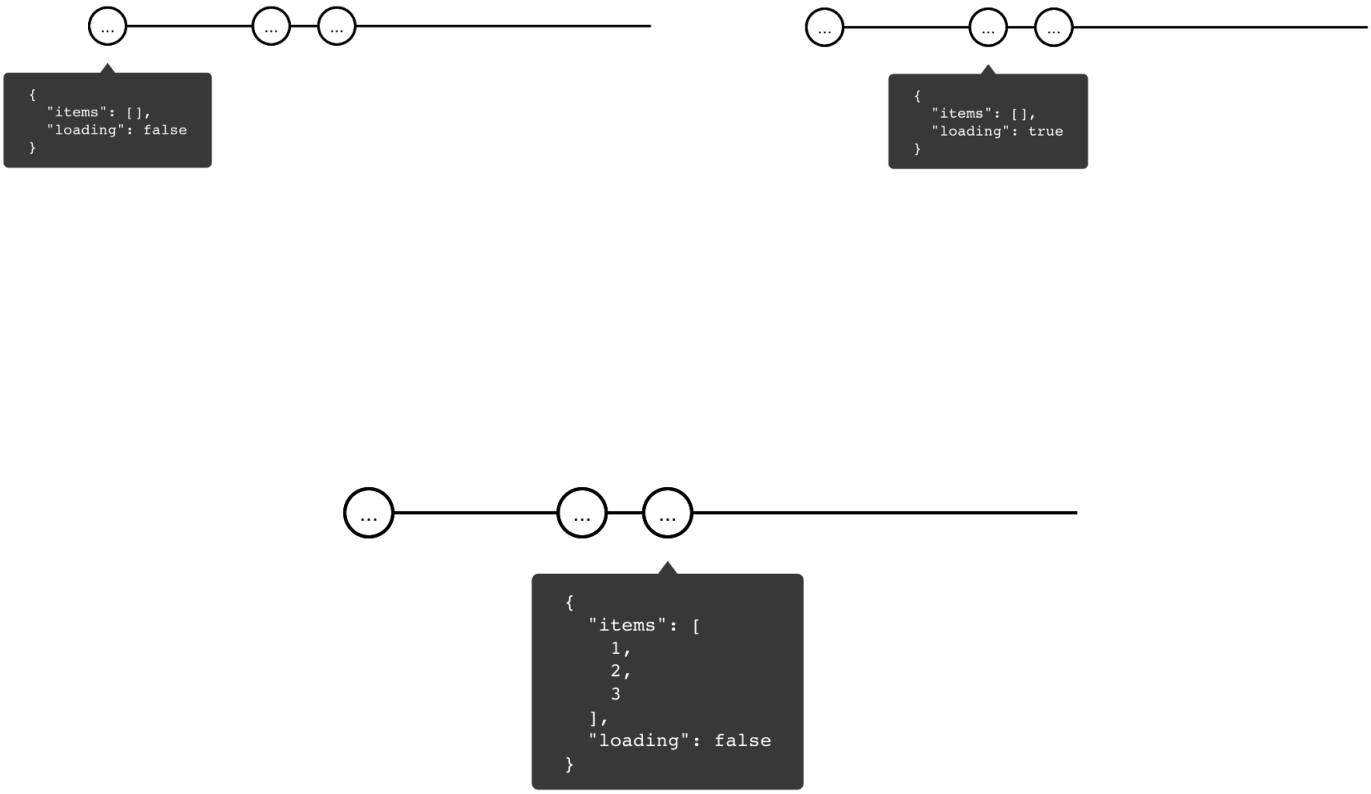
```
dispatcher$: Observable<Action> = merge(actions$,
load$);
```



https://rxviz.com/v/38l61KEO

To put everything together, we're replacing **actions$** stream with **dispatcher$**.

```
function stateReducer(state, action) {
  switch (action.type) {
    // ...
    case 'load success':
      return {
        ...state,
        items: action.payload,
        loading: false
      };
    // ...
  }
}

state$: Observable<ItemsState> =
  dispatcher$.pipe(
    startWith(defaultState),
    scan(stateReducer),
    publishReplay(1),
    refCount()
  );
```



https://rxviz.com/v/38jdvK9O

. . .

Error processing

And one more important point is the correct error handling. Let's make a request that will continually return an error. To do this, create a new function **loadWithError()**, which will emulate an error when loading with the same delay of 1 second.

timer()

timer() starts the execution of the stream after the specified time, in our case after 1 second.

switchMapTo()

switchMapTo() does a switch to the stream, in our case we simply return the stream with an error.

throwError()

throwError() creates a stream with an error.

```
function loadWithError() {
  return timer(1000).pipe(
    switchMapTo(throwError('Something wrong!'))
  );
}
```

Let's hook it into our **load$** effect, and use the **catchError()** operator for error handling.

catchError()

catchError() is triggered if the stream **completes** with an error and allows it to be processed.

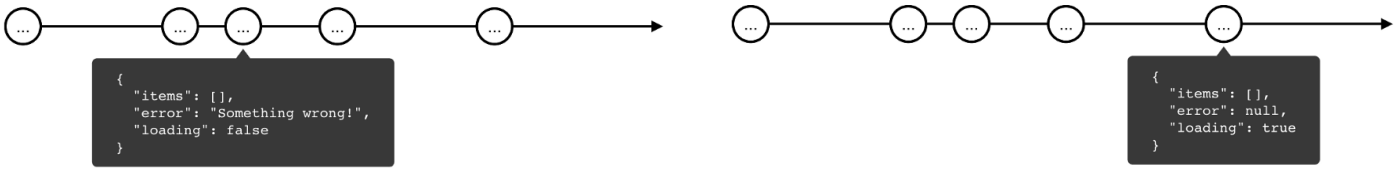
```
/**
 * Wrong code (!)
 **/


const load$ =
  actions$.pipe(
    ofType('load'),
    switchMap(() => loadWithError()),
    map((data) => ({
      type: 'load success',
      payload: data
    })),
    catchError((error) => of({
      type: 'load failed',
      payload: error
    }))
  );
```

```
/**
 * Wrong code (!)
 **/
```

And we will process the received command with an error in our **stateReducer()**. Note that after load initialization we reset the error.

```
function stateReducer(state, action) {
  switch (action.type) {
    case 'load':
      return {
```

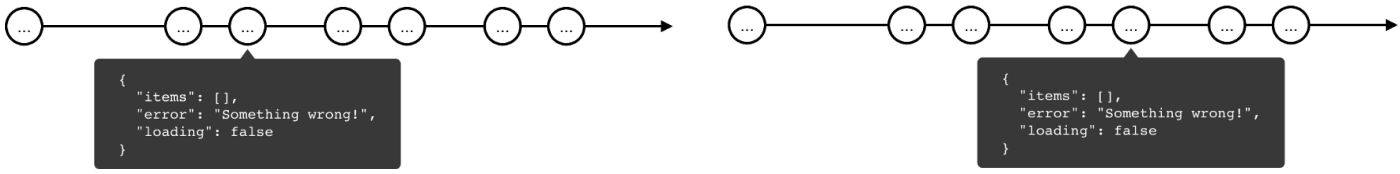
```
          ...state,
          error: null,
          loading: true
        };
      // ...
      case 'load failed':
        return {
          ...state,
          error: action.payload,
          loading: false
        };
      // ...
    }
}
```



https://rxviz.com/v/7Ja55l0J

As you can see, the effect works **only once**, although three commands are sent to download. This is due to the fact that the flow with the effect of **load$** ends and no longer receives commands. Let's fix it. To do this, we need to transfer the processing of data load and error handling under **switchMap()**.

```
const load$ =
  actions$.pipe(
    ofType('load'),
    switchMap(() =>
      loadWithError().pipe(
        map((data) => ({
          type: 'load success',
          payload: data
        })),
        catchError((error) => of({
          type: 'load failed',
          payload: error
        }))
      )
    )
  );
```



https://rxviz.com/v/7J244eao

Now our errors are processed correctly, and the flow with the effect does not end after errors. Cheers!

. . .

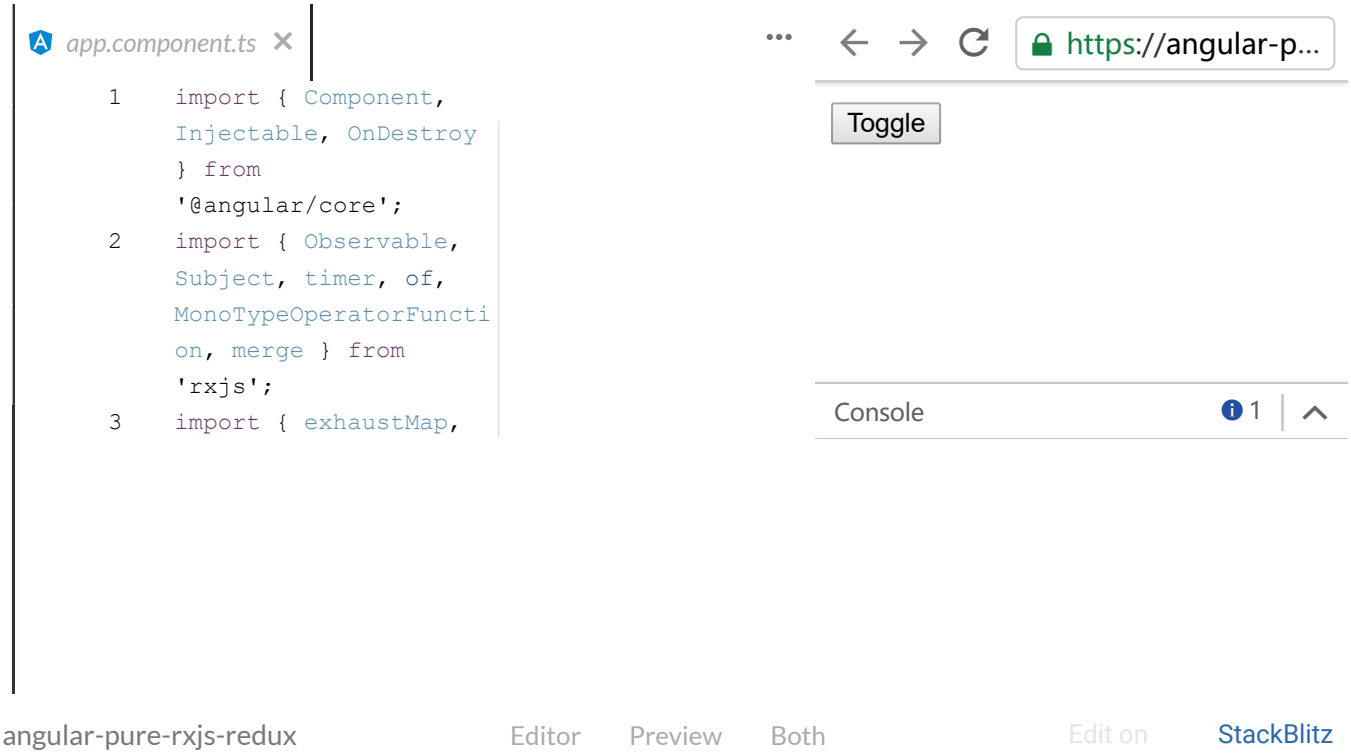
Conclusion

This is not a production-ready solution but even in the current state, it provides much more freedom than existing tools!

For **RxJS** newbies, try other operators with this solution or writing your own, **select()** for example.

Also, note that every screenshot in this post has the link to **rxviz.com**, RxJS playground.

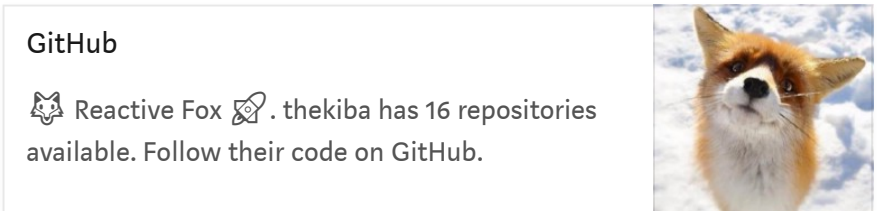
A complete solution on **stackblitz.com**.



. . .

You can always contact me in the **telegram**.

And subscribe to my **GitHub** and **Medium**.



github.com

Вертихвост キバ

Reactive Fox

t.me