Epics · redux-observable

Epics

> **Not familiar with Observables/RxJS v5?**
>
> redux-observable requires an understanding of Observables with RxJS v5. If you're new to Reactive Programming with RxJS v5, head over to http://reactivex.io/rxjs/ to familiarize yourself first.
>
> redux-observable (because of RxJS) truly shines the most for complex async/side effects. While most people will use redux-observable all of them, if you're not already comfortable with RxJS you might consider using redux-thunk for simple side effects and then use redux-observable for the complex stuff. That way you can remain productive and learn RxJS as you go.

An **Epic** is the core primitive of redux-observable.

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

You can think of it of having roughly this type signature:

```
function (action$: Observable<Action>, store: Store): Observable<Action>;
```

While you'll most commonly produce actions out in response to some action you received in, keep in mind that's not actually a requirement! Once you're inside your Epic, use any Observable patterns you desire as long as anything output from the outermost stream is an action.

The actions you emit will be immediately dispatched through the normal `store.dispatch()`, so under the hood redux-observable effectively does `epic(actions$, store).subscribe(store.dispatch)`

> The pattern of handling side effects this way is similar to the "*process manager*" pattern, sometimes called a "*saga*", but the original definition of saga is not truly applicable. If you're familiar with redux-saga, redux-observable is very similar. But because it uses RxJS it is much more declarative and you utilize and expand your existing RxJS abilities.

Keep in mind that letting an incoming action pass through transparently will create an infinite loop:

```
// DO NOT DO THIS
const actionEpic = (action$) => action$; // creates infinite loop
```

A Basic Example

> **IMPORTANT:** redux-observable does not add any of the RxJS operators to the `Observable.prototype` so you will need to import the ones you use or import all of them in your entry file.
>
> Because there are many ways to add them, our examples will not include any imports. If you want to add every operator, put `import 'rxjs';` in your entry `index.js`. Learn more.

Let's start with a simple Epic example:

```
const pingEpic = action$ =>
  action$.filter(action => action.type === 'PING')
    .mapTo({ type: 'PONG' });

// later...
dispatch({ type: 'PING' });
```

> Noticed how `action$` has a dollar sign at the end? It's simply a common RxJS convention to identify variables that reference a stream.

`pingEpic` will listen for actions of type `PING` and map them to a new action, `PONG`. This example is functionally equivalent to doing this:

```
dispatch({ type: 'PING' });
dispatch({ type: 'PONG' });
```

> Epics run alongside the normal Redux dispatch channel, after the reducers have received them, so you cannot "swallow" an incoming action. When you map one of those actions to another, you're not preventing the original action from reaching the reducers; that action has already been through them!

The real power starts to reveal itself when you need to do something asynchronous. Let's say you want to dispatch `PONG` 1 second after receiving the `PING`:

```
const pingEpic = action$ =>
  action$.filter(action => action.type === 'PING')
    .delay(1000) // Asynchronously wait 1000ms then continue
    .mapTo({ type: 'PONG' });

// later...
dispatch({ type: 'PING' });
```

Your reducers will receive the original `PING` action, then 1 second later receive the `PONG`.

```
const pingReducer = (state = { isPinging: false }, action) => {
  switch (action.type) {
    case 'PING':
      return { isPinging: true };

    case 'PONG':
      return { isPinging: false };

    default:
      return state;
  }
};
```

Since filtering by a specific action type is so common, the actions stream redux-observable gives you is a special `ActionsObservable` which has a custom `ofType()` operator to reduce that boilerplate:

```
const pingEpic = action$ =>
  action$.ofType('PING')
    .delay(1000) // Asynchronously wait 1000ms then continue
    .mapTo({ type: 'PONG' });
```

Try It Live!

A Real World Example

Now that we have a general idea of what an Epic looks like, let's continue with a more real-world example:

```
// action creators
const fetchUser = username => ({ type: FETCH_USER, payload: username });
const fetchUserFulfilled = payload => ({ type: FETCH_USER_FULFILLED, payload });

// epic
const fetchUserEpic = action$ =>
  action$.ofType(FETCH_USER)
    .mergeMap(action =>
      ajax.getJSON(`https://api.github.com/users/${action.payload}`)
        .map(fetchUserFulfilled)
    );
```

```
// later...
dispatch(fetchUser('torvalds'));
```

> We're using action creators (aka factories) like `fetchUser` instead of creating the action POJO directly. This is a Redux convention that is totally optional.

We have a standard Redux action creator `fetchUser`, but also a corresponding Epic to orchestrate the actual AJAX call. When that AJAX call comes back, we map the response to a `FETCH_USER_FULFILLED` action.

> Remember, Epics take stream of **actions in** and return a stream of **actions out**. If the RxJS operators and behavior shown so far is unfamiliar to you, you'll definitely want to take some time to dive deeper into RxJS before proceeding.

You can then update your Store's state in response to that `FETCH_USER_FULFILLED` action:

```
const users = (state = {}, action) => {
  switch (action.type) {
    case FETCH_USER_FULFILLED:
      return {
        ...state,
        // `login` is the username
        [action.payload.login]: action.payload
      };

    default:
      return state;
  }
};
```

Try It Live!

Accessing the Store's State

Your Epics receive a second argument, the Redux store itself.

```
function (action$: Observable<Action>, store: Store): Observable<Action>;
```

With this, you can call `store.getState()` to synchronously access the current state:

```
const INCREMENT = 'INCREMENT';
```

```
const INCREMENT_IF_ODD = 'INCREMENT_IF_ODD';

const increment = () => ({ type: INCREMENT });
const incrementIfOdd = () => ({ type: INCREMENT_IF_ODD });

const incrementIfOddEpic = (action$, store) =>
  action$.ofType(INCREMENT_IF_ODD)
    .filter(() => store.getState().counter % 2 === 0)
    .map(increment);

// later...
dispatch(incrementIfOdd());
```

> When an Epic receives an action, it has already been run through your reducers and the state updated, if needed.

Remember, `store.getState()` is just an imperative, synchronous API. You cannot treat it as a stream as-is.

Try It Live!

Combining Epics

Finally, redux-observable provides a utility called `combineEpics()` that allows you to easily combine multiple Epics into a single one:

```
import { combineEpics } from 'redux-observable';

const rootEpic = combineEpics(
  pingEpic,
  fetchUserEpic
);
```

Note that this is equivalent to:

```
import { merge } from 'rxjs/observable/merge';

const rootEpic = (action$, store) => merge(
  pingEpic(action$, store),
  fetchUserEpic(action$, store)
);
```

Next Steps

Next, we'll explore how to activate your Epics so they can start listening for actions.