Managing side effects in React with React RocketJump

A
Alberto Osio Follow
Jul 10 · 4 min read

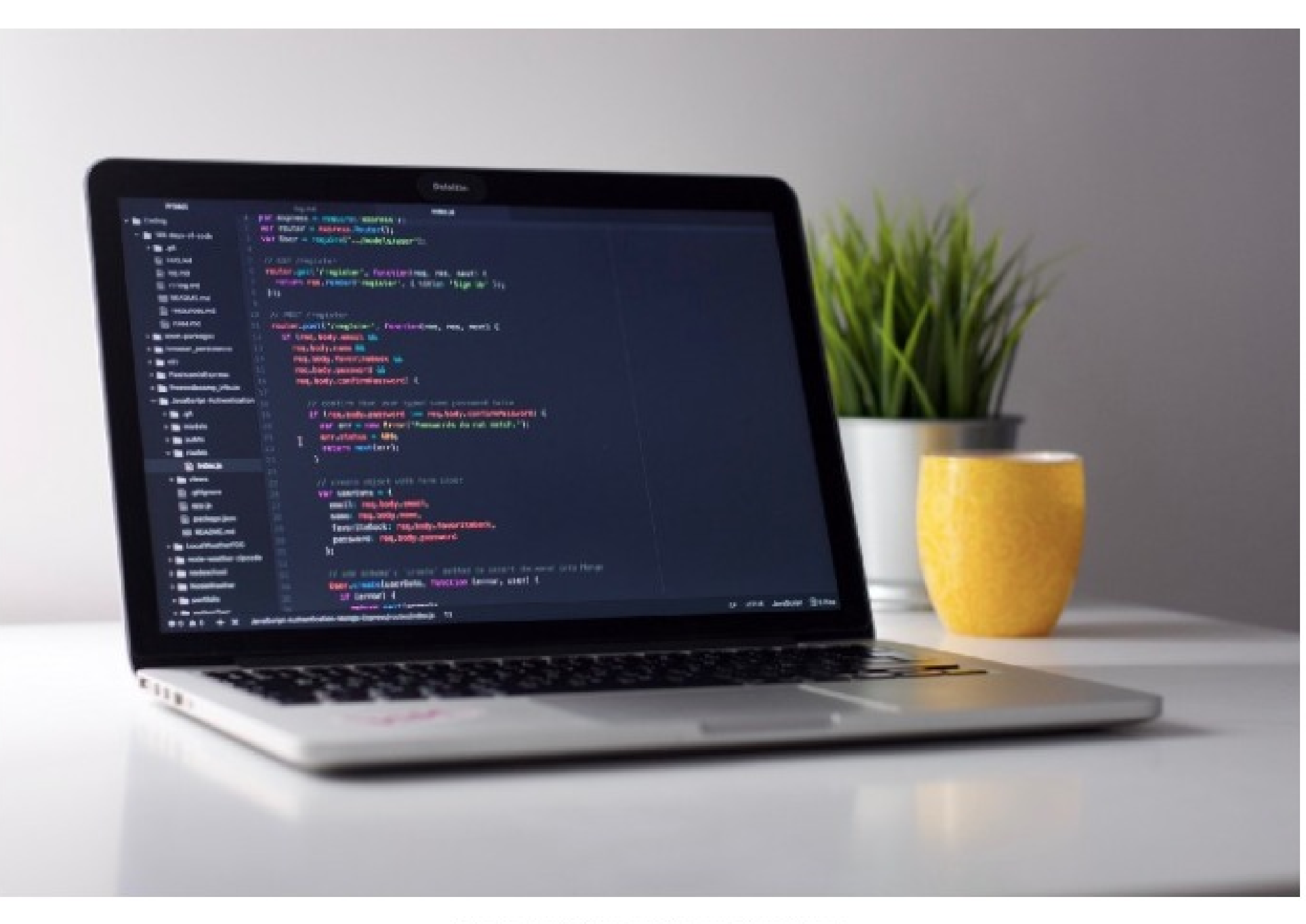

Photo by Clément H on Unsplash

React is really a great tool when you have to build complex user interfaces, but still it plods when we come at side effects and asynchronous operations in general. Many libraries exist to help dealing with this problem, but they are complex and usually rely on some context based API (maybe in disguise, but hey, indeed it is a context). Using context can have some advantages, but definitely it as a great disadvantage: it breaks component encapsulation and isolation, ending up in breaking reusability. In React reusability is a core (and highly desirable) feature, so why giving up so easily?

Today there is a new tool to manage side effects and asynchronous operations in React: React-RocketJump

React RocketJump: manage side effects like a breeze

One of the core objectives in building React RocketJump was to get to something simple to use and quick to write, with almost no dependency except for React.

Ok, let's see it in action: sometimes, examples tell more than words

First of all, take some API you want to use in your application. For sake of generality, I'll use *https://my.local.host/api/v1/movies*, which is supposed to return a list of movies in JSON format. Suppose it is not authenticated (don't worry, React RocketJump supports it, but keep it simple)

Now we need to create a *RocketJump Object*, which is a sort of description of our side effect and of how we want the library to manage it.

```
export const MoviesState = rj({
    effect: () => fetch('https://my.local.host/api/v1/movies')
                      .then(response => response.json())
})
```

index.jsx hosted with ❤ by GitHub view raw

Ok, this is enough for a basic description. We are telling React RocketJump to create a *RocketJump Object* wrapping an asynchronous task. This async task fetches the URL we spoke about beforehand, parses the response as JSON and returns it.

You can use any library of your choice to make requests, even the good old XMLHttpRequest, provided that you return a Promise (or a Rx observable, but this is another story) from the *effect*.

Now it's time to create a component that uses the *RocketJump Object* to interact with the API

```
import React from 'react'

const MyMoviesList = () => {
    const movies = [] // This is where React RocketJump will come in

    return (
        <table>
            <thead>
                <tr>Title</tr>
                <tr>Duration</tr>
            </thead>
            <tbody>
                {movies.map(movie => (
                    <tr key={movie.id}>
                        <td>{movie.title}</td>
                        <td>{movie.duration}</td>
                    </tr>
                ))}
            </tbody>
        </table>
    )
}
```

Snippet1.jsx hosted with ❤ by GitHub view raw

Now, let's focus on the RocketJump part. React RocketJump provides different ways to connect *RocketJump Object* instances and Components. Here we will use *hooks*, and in particular the *useRj* hook

This hooks takes a *RocketJump Object* and returns two elements: a *state* and an *action bag*.

The *state* is the state of our task, and contains the following properties

- *pending*: this tells us if our task is running (eg if we are awaiting for a response)
- *data*: the resolved value of the most recent task execution (the effect function returned a Promise, do you remember?)
- *error*: the last error triggered by the task

The *action bag* instead contains some control switches for the task:

- *run* triggers the task
- *cancel* cancels a pending run
- *clean* cancels any pending run and resets the state object to the default value

Given this, the interesting parts (for us) are *data* and *pending* in the *state*, and *run* in the *action bag*

Now, put things together

```
import React from 'react'
import { useRj } from 'react-rocketjump'
import { MoviesState } from './states'

const MyMoviesList = () => {
    const [state, actions] = useRj(MoviesState)

    const isLoading = state.pending
    const movies = state.data
    const loadMovies = actions.run

    if (movies === null || isLoading) {
        return <div>Loading</div>
    }

    return (
        <table>
            <thead>
                <tr>Title</tr>
                <tr>Duration</tr>
            </thead>
            <tbody>
                {movies.map(movie => (
                    <tr key={movie.id}>
                        <td>{movie.title}</td>
                        <td>{movie.duration}</td>
                    </tr>
                ))}
            </tbody>
        </table>
    )
}
```

index.jsx hosted with ❤ by GitHub view raw

We have only one step missing: trigger our task (and query our API) and populate consequently the *movies* variable

We can use *useEffect* core hook for this

```
import React, { useEffect } from 'react'
import { useRj } from 'react-rocketjump'
import { MoviesState } from './states'

const MyMoviesList = () => {
    const [state, actions] = useRj(MoviesState)

    const isLoading = state.pending
    const movies = state.data
    const loadMovies = actions.run

    useEffect(() => {
        loadMovies()
    }, [loadMovies])

    if (movies === null || isLoading) {
        return <div>Loading</div>
    }

    return (
        <table>
            <thead>
                <tr>Title</tr>
                <tr>Duration</tr>
            </thead>
            <tbody>
                {movies.map(movie => (
                    <tr key={movie.id}>
                        <td>{movie.title}</td>
                        <td>{movie.duration}</td>
                    </tr>
                ))}
            </tbody>
        </table>
    )
}
```

index.jsx hosted with ❤ by GitHub view raw

Yes, we are done!

How this works?

Well, we described an asynchronous operation, whose implementation was the fetching of some data from an API. Then, we connected this task to a component using an appropriate hook. Inside the component, we extracted meaningful items from the hook return value. In the first render of the component, *isLoading* will be false since we never triggered the task, and *movies* will be null since we have no "last" execution to return data of (this explains the need of the *if* with the loading message). After the first render, the *useEffect* is fired and the task triggered by means of the *run* function (that we assigned to the *loadMovies* constant in our example). This will cause a second render of the component, in which *movies* is still *null*, and *isLoading* will be set to *true*. Finally, once the promise resolves, a third render pass will be triggered, with *movies* set to API result and *isLoading* set to *false*.

Which is the difference with plain data loading?

There are a number of reasons to consider switching to React RocketJump for side effects management. The main ones are:

1. It allows to decouple the definition of async tasks (primarily, fetching from API endpoints) from the components that need those data
2. *It integrates with component lifecycle, so that you don't have to consider the problem of setting state on unmounted components, which is always around when dealing with async operations in React*
3. The React RocketJump has a number of features which can greatly help in many many pratical cases. Some examples? Debouncing calls, dealing with pagination, managing lists, deal with concurrent operations, …
4. React RocketJump is built with composability and reusability in mind, so you will be able to write cleaner, easier and shorter code

Want to learn more? Stay tuned! In our next article, we will deal with debouncing requests

Can't wait? Head to the docs and give it a try!

See you!

React Side Effects Reactjs API JavaScript

7 claps

A
WRITTEN BY
Alberto Osio
Follow

Write the first response