Published in Better Programming

Mohammad Faisal Follow
Mar 29, 2021 · 3 min read ★ · Listen

How To Apply SOLID Principles To Clean Your Code in React

A look at the single-responsibility principle (SRP) in action

Photo by MockupEditor.com from Pexels.

Today, we will start with a bad code example, apply the first principle of SOLID, and see how this can help us to write small, beautiful, and clean React components with clear responsibilities.

Let's get started.

What Is the Single-Responsibility Principle?

What the single-responsibility principle tells us is that each class or component should have a single purpose of existence.

Components should do only one thing and do that well.

Let's refactor a bad but working piece of code and make it cleaner and better by using this principle.

Let's Start With a Bad Example

Let's first see some code that violates this principle. The comments are added for better understanding:

```
import React, {useEffect, useReducer, useState} from "react";

const initialState = {
    isLoading: true
};

// COMPLEX STATE MANAGEMENT
function reducer(state, action) {
    switch (action.type) {
        case 'LOADING':
            return {isLoading: true};
        case 'FINISHED':
            return {isLoading: false};
        default:
```




```
20
21      const [users , setUsers] = useState([])
22      const [filteredUsers , setFilteredUsers] = useState([])
23      const [state, dispatch] = useReducer(reducer, initialState);
24
25      const showDetails = (userId) => {
26          const user = filteredUsers.find(user => user.id===userId);
27          alert(user.contact)
28      }
29
30      // REMOTE DATA FETCHING
31      useEffect(() => {
32          dispatch({type:'LOADING'})
33          fetch('https://jsonplaceholder.typicode.com/users')
34              .then(response => response.json())
35              .then(json => {
36                  dispatch({type:'FINISHED'})
37                  setUsers(json)
38              })
39      },[])
40
41      // PROCESSING DATA
42      useEffect(() => {
43          const filteredUsers = users.map(user => {
44              return {
45                  id: user.id,
46                  name: user.name,
47                  contact: `${user.phone} , ${user.email}`
48              };
49          });
50          setFilteredUsers(filteredUsers)
51      },[users])
52
53      // COMPLEX UI RENDERING
54      return <>
55          <div> Users List</div>
56          <div> Loading state: {state.isLoading? 'Loading': 'Success'}</div>
57          {users.map(user => {
58              return <div key={user.id} onClick={() => showDetails(user.id)}>
59                  <div>{user.name}</div>
60                  <div>{user.email}</div>
61              </div>
```




What this code does

This is a functional component where we fetch data from a remote source, filter the data, and then show it in the UI. We also detect the loading state of the API call.

I have kept this example short for better understanding. But you can find these in the same component almost anywhere! There are a lot of things going on here:

1. Remote data fetching
2. Data filtering
3. Complex state management
4. Complex UI functionality

So let's explore how we can improve the design of the code and make it tight.

1. Move Data Processing Logic Out

Never keep your HTTP calls inside the component. It's a rule of thumb. There are several strategies you can follow to remove these codes from the component.

The least you should do is create a custom Hook and move your data-fetching logic there. For example, we can create a Hook named `useGetRemoteData` that looks like this:

```
1  import {useEffect, useReducer, useState} from "react";
2
3  const initialState = {
4      isLoading: true
5  };
6
7  function reducer(state, action) {
8      switch (action.type) {
9          case 'LOADING':
10             return {isLoading: true};
11         case 'FINISHED':
12             return {isLoading: false};
```




```
export const useGetRemoteData = (url) => {

    const [users , setUsers] = useState([])
    const [state, dispatch] = useReducer(reducer, initialState);

    const [filteredUsers , setFilteredUsers] = useState([])


    useEffect(() => {
        dispatch({type:'LOADING'})
        fetch('https://jsonplaceholder.typicode.com/users')
            .then(response => response.json())
            .then(json => {
                dispatch({type:'FINISHED'})
                setUsers(json)
            })
    },[])

    useEffect(() => {
        const filteredUsers = users.map(user => {
            return {
                id: user.id,
                name: user.name,
                contact: `${user.phone} , ${user.email}`
            };
        });
        setFilteredUsers(filteredUsers)
    },[users])

    return {filteredUsers , isLoading: state.isLoading}
}
```

Now our main component will look like this:

```
import React from "react";
import {useGetRemoteData} from "./useGetRemoteData";

export const SingleResponsibilityPrinciple = () => {

    const {filteredUsers , isLoading} = useGetRemoteData()
```

```
12
13      return <>
14          <div> Users List</div>
15          <div> Loading state: {isLoading? 'Loading': 'Success'}</div>
16          {filteredUsers.map(user => {
17              return <div key={user.id} onClick={() => showDetails(user.id)}>
18                  <div>{user.name}</div>
19                  <div>{user.email}</div>
20              </div>
21          })}
22      </>
23  }
```

FirstRefactor.is hosted with ❤️ by GitHub view raw

FirstRefactor.js

Look how our component is now smaller and easier to understand! This is the simplest and first thing you can do in a convoluted codebase.

But we can do better.

2. Reusable Hook for Data Fetching

Now when we see our `useGetRemoteData` Hook, we see that this Hook is doing two things:

1. Fetching data from a remote source
2. Filtering data

Let's extract the logic of fetching remote data to a separate Hook named `useHttpGetRequest` that takes the URL as a component:

```
1   import {useEffect, useReducer, useState} from "react";
2   import {loadingReducer} from "./LoadingReducer";
3
4   const initialState = {
5       isLoading: true
```




```
11      const [state, dispatch] = useReducer(loadingReducer, initialState);
12
13      useEffect(() => {
14          dispatch({type:'LOADING'})
15          fetch(URL)
16              .then(response => response.json())
17              .then(json => {
18                  dispatch({type:'FINISHED'})
19                  setUsers(json)
20              })
21      },[])
22
23      return {users , isLoading: state.isLoading}
24
25  }
```


useHttpgetRequest.js hosted with ❤️ by GitHub view raw

useHttpGetrequest.js

We also removed the reducer logic to a separate file:

```
1   export function loadingReducer(state, action) {
2       switch (action.type) {
3           case 'LOADING':
4               return {isLoading: true};
5           case 'FINISHED':
6               return {isLoading: false};
7           default:
8               return state;
9       }
10  }
```

LoadingReducer.js hosted with ❤️ by GitHub view raw

LoadingReducer.js

So now our `useGetRemoteData` becomes:

```
1   import {useEffect, useState} from "react";
2   import {useHttpGetRequest} from "./useHttpGet";
3   const REMOTE_URL = 'https://jsonplaceholder.typicode.com/users'
4
```




```
10          const filteredUsers = users.map(user => {
11              return {
12                  id: user.id,
13                  name: user.name,
14                  contact: `${user.phone} , ${user.email}`
15              };
16          });
17          setFilteredUsers(filteredUsers)
18      },[users])
19
20      return {filteredUsers , isLoading}
21  }
```

useGetRemoteData.js hosted with ❤️ by GitHub view raw

useGetRemoteData.js

Much cleaner, right? Can we do better? Sure, why not?

3. Decompose UI Components

Take a look at our component where we show the details of a user. We can create a reusable `UserDetails` component for that purpose:

```
1   const UserDetails = (user) => {
2
3       const showDetails = (user) => {
4           alert(user.contact)
5       }
6
7       return <div key={user.id} onClick={() => showDetails(user)}>
8           <div>{user.name}</div>
9           <div>{user.email}</div>
10      </div>
11  }
```

UserDetai.s.js hosted with ❤️ by GitHub view raw

UserDetails.js

Finally, our original component becomes:

```
export const Users = () => {
    const {filteredUsers , isLoading} = useGetRemoteData()

    return <>
        <div> Users List</div>
        <div> Loading state: {isLoading? 'Loading': 'Success'}</div>
        {filteredUsers.map(user => <UserDetails user={user}/>)}
    </>
}
```

Users.js hosted with ❤️ by GitHub view raw

Users.js

We slimmed our code down from 60 lines to 12 lines! And we created five separate components, each with a clear and single responsibility.

Let's Review What We Just Did

Let's review our components and see if we achieved the SRP:

- `Users.js` — Responsible for displaying the user list
- `UserDetails.js` — Responsible for displaying details of a user
- `useGetRemoteData.js` — Responsible for filtering remote data
- `useHttpGetrequest.js` — Responsible for HTTP calls
- `LoadingReducer.js` — Complex state management

Of course, we can improve a lot of other things, but this should work as a good starting point for you.

Conclusion

This was a simple demonstration of how you can reduce the amount of code in each file

1. Open Closed Principle
2. Liskov Substitution Principle
3. Interface Segregation Principle
4. Dependency Inversion Principle

Get in touch with me via LinkedIn or my Personal Website.