

# Single Responsibility Principle in React Applications

FOLLOW THE SINGLE RESPONSIBILITY  
PRINCIPLE TO BUILD AN UNDERSTANDABLE  
AND CHANGE-READY REACT CODEBASE



**sunscrapers**

# Single Responsibility Principle in React Applications

If you work in React projects, you've probably seen this:

**Long blocks of code that use many variables and allow complex execution paths. In short, React components doing way more than they should.**

Putting together pieces of code that perform distinct tasks is a problem because it inhibits code reusability, makes testing difficult, and increases the time required to understand the code.

**Solution? The Single Responsibility Principle.**

Read this ebook to see how following the Single Responsibility Principle allows building a React codebase that is easy to understand and ready for changes. To show you how it works in practice, we are going to refactor a fat React component into smaller pieces, each with its very own, single responsibility.



**sunscrapers**

# Table of contents

Introduction	4
Why the Single Responsibility Principle?	4
<b>Chapter 1:</b> Refactoring a React component	5
Component - main tasks	6
Component - responsibilities	6
<b>Chapter 2:</b> Extracting the data fetching functionality	7
Summary	9
<b>Chapter 3:</b> Refactoring the data fetching service	10
Summary	14
<b>Chapter 4:</b> Freeing the component from data fetching management	15
Summary	24
Conclusion	25



AUTHOR

**Jacek Mikrut**

Jacek is a senior front-end developer at Sunscrapers. He earned an MSc degree in Computer Science from the Silesian University of Technology and has since worked as a full-stack developer for a number of companies, today focusing on front-end applications. Passionate about best practices in software development processes and opportunities offered by the newest web application technologies.

EDITOR

**Anahita Rouyan**



# Introduction

In my work as a front-end developer, I often encounter single methods, classes, and React components doing way too much.

These blocks of code are typically long, use many variables, and allow complex execution paths. Pieces of code performing distinct tasks are often coupled together. That inhibits code reusability, makes testing cumbersome, and increases the time required to understand the code.

That's where the Single Responsibility Principle comes in handy.



## Why the Single Responsibility Principle?

The Single Responsibility Principle simply means that a method, a class, or a module has only one responsibility.

For example:

The responsibility of a method may be adding two numbers. The responsibility of a class may be providing a calculator interface. The responsibility of a module may be connecting its internal parts together.


Applying the Single Responsibility Principle often leads to developing smaller pieces of code where each is focused on just one task. Then we can have these pieces cooperate together to perform more complex operations.

Remember that breaking code into small pieces isn't always the best way to go. It depends on the project and specific circumstances. Sometimes making things reusable brings no benefits at all - for instance, in a small project or script where we just know that we won't be reusing any of its elements. That's why before following the Single Responsibility Principle we need to decide what pays off in particular circumstances.

Yet, in most software projects keeping the Single Responsibility Principle is crucial.

Why? Because it reduces time and cost of introducing changes, bug fixes, and new functionalities.

**In this ebook, I will go through an example of refactoring a fat React component into smaller pieces: a service, utility functions, and other components - and comment on the benefits each step of that process brings.**

 *Note: This ebook series assumes that readers have some knowledge of React, including the higher-order components, and ECMAScript 2015 features like arrow functions, classes, promises, and symbols.*

# #1 Why the Single Responsibility Principle?

Let's start right away with the code. Have a look at the code below and check how much time you need to understand what this component does.

// src/components/LatestReactPullRequest/LatestReactPullRequest.js

```

1  import React, { Component, Fragment } from 'react';
2
3  import moment from 'moment';
4  import makePromiseCancelable from '@utils/makePromiseCancelable';
5  import './LatestReactPullRequest.css';
6
7  const DATA_FETCHING_STATUS = {
8    NOT_STARTED: Symbol('DATA_FETCHING_NOT_STARTED'),
9    IN_PROGRESS: Symbol('DATA_FETCHING_STATUS_IN_PROGRESS'),
10   SUCCESS:     Symbol('DATA_FETCHING_STATUS_SUCCESS'),
11   FAILURE:     Symbol('DATA_FETCHING_STATUS_FAILURE'),
12 };
13
14 class LatestReactPullRequest extends Component {
15   constructor(props) {
16     super(props);
17     this.state = {
18       dataFetchingStatus: DATA_FETCHING_STATUS.NOT_STARTED,
19       data: null,
20     };
21   }
22
23   componentDidMount() {
24     this.setState({
25       dataFetchingStatus: DATA_FETCHING_STATUS.IN_PROGRESS,
26     });
27
28     const fetchPromise = fetch('https://api.github.com/repos/facebook/react/pulls?state=all&sort=created&direction=desc&per_page=1&page=1', {
29       headers: {
30         'Accept': 'application/vnd.github.v3+json',
31       }
32     });
33
34     const { promise, cancel } = makePromiseCancelable(fetchPromise);
35     this.cancelDataFetchingPromise = cancel;
36
37     promise
38       .then((response) => {
39         if (response.status !== 200) {
40           throw new Error(`Response status code: ${response.status}`);
41         } else {
42           response.json()
43             .then(([latestReactPullRequest]) => {
44               this.setState({
45                 dataFetchingStatus: DATA_FETCHING_STATUS.SUCCESS,
46                 data: {
47                   title: latestReactPullRequest.title,
48                   body: latestReactPullRequest.body,
49                   userLogin: latestReactPullRequest.user.login,
50                   createdAt: latestReactPullRequest.created_at,
51                 }
47               });
52             });
53           });
54         }
55       })
56       .catch((error) => {
57         if (error.isCanceled) return;
58
59         this.setState({
60           dataFetchingStatus: DATA_FETCHING_STATUS.FAILURE,
61           data: null,
62         });
63       });
64   }
65

```

```

66   componentWillUnmount() {
67     this.cancelDataFetchingPromise();
68   }
69
70   render() {
71     const { dataFetchingStatus, data } = this.state;
72     return (
73       <div className="c-latest-react-pull-request">
74         {dataFetchingStatus === DATA_FETCHING_STATUS.NOT_STARTED && (
75           <Fragment>Initializing...</Fragment>)
76         }
77         {dataFetchingStatus === DATA_FETCHING_STATUS.IN_PROGRESS && (
78           <Fragment>Fetching...</Fragment>
79         )}
80         {dataFetchingStatus === DATA_FETCHING_STATUS.FAILURE && (
81           <Fragment>Data fetching error...</Fragment>
82         )}
83         {dataFetchingStatus === DATA_FETCHING_STATUS.SUCCESS && (
84           <Fragment>
85             <div className="c-latest-react-pull-request__title">{data.title}</div>
86             <div className="c-latest-react-pull-request__body">{data.body}</div>
87             <div className="c-latest-react-pull-request__created-at-and-user-login">
88               {moment(data.createdAt).calendar()} by {data.userLogin}
89             </div>
90           </Fragment>
91         )}
92       </div>
93     );
94   }
95 }
96
97 export default LatestReactPullRequest;

```

## Component - main tasks

The component's name reveals that it relates to the latest React pull request.

**Let's list its main tasks. The component:**

1. Fetches information about the most recently created pull request in React repo on GitHub,
2. Shows the current fetching status,
3. Displays the title, body, creation time, and author's login of the pull request.

The component looks self-sufficient in performing these three tasks. That may be an acceptable solution if none of these functionalities are duplicated in other places in the application and we don't intend to change this code later.

Note that in a typical front-end application, we would probably need to reuse some of this functionality in other components.

## Component - responsibilities

Let's take a look at the component's code again and see what responsibilities it has. Our component is responsible for:

1. ... handling the network request,
2. ... knowing what the URL to the React repo is,
3. ... knowing how to ask the GitHub API for the latest pull request,
4. ... treating a non-200 HTTP response to "failure",
5. ... extracting required data from the response,
6. ... monitoring the status of the fetching operation,
7. ... displaying the fetching status,
8. ... rendering the latest pull request data.

Let's distribute these responsibilities and observe what benefits it brings.

## #2 Extracting the data fetching functionality

We start with extracting the data fetching functionality to a service.

First, we need to prepare the code for extraction. Right now, our code is mixed with the component's `setState` operations.

We can achieve that by restructuring the operations in the promise chain into two stages:

- In the first stage, data is extracted from the response, packed to an object, and passed down (this code is general

and decoupled from the surrounding component, it doesn't know how the data will be consumed),

- In the second stage, the data and fetching status are saved to the component's state (this code is bound to the component and decoupled from the data source, i.e. it doesn't know where the data comes from).

// src/components/LatestReactPullRequest/LatestReactPullRequest.js

```

1 // ...
2
3 componentDidMount() {
4   this.setState({
5     dataFetchingStatus: DATA_FETCHING_STATUS.IN_PROGRESS,
6   });
7
8   // stage 1
9
10  const fetchPromise = fetch('https://api.github.com/repos/facebook/react/pulls?state=all&sort=created&direction=desc&per_page=1&page=1', {
11    headers: {
12      'Accept': 'application/vnd.github.v3+json',
13    }
14  })
15  .then(function(response) {
16    if (response.status !== 200) {
17      throw new Error(`Response status code: ${response.status}`);
18    } else {
19      return response.json()
20        .then(function([latestReactPullRequest]) {
21          return {
22            title: latestReactPullRequest.title,
23            body: latestReactPullRequest.body,
24            userLogin: latestReactPullRequest.user.login,
25            createdAt: latestReactPullRequest.created_at,
26          }
27        });
28    }
29  });
30
31  // stage 2
32
33  const { promise, cancel } = makePromiseCancelable(fetchPromise);
34  this.cancelDataFetchingPromise = cancel;
35
36  promise
37  .then((data) => {
38    this.setState({
39      dataFetchingStatus: DATA_FETCHING_STATUS.SUCCESS,
40      data
41    });
42  })
43  .catch((error) => {
44    if (error.isCanceled) return;
45
46    this.setState({
47      dataFetchingStatus: DATA_FETCHING_STATUS.FAILURE,
48      data: null,
49    });
50  });
51 }
52
53 // ...

```

Since the code in Stage 1 is now component-independent, we can extract it to a service.

#### // src/services/github/actions/fetchLatestReactPullRequest/fetchLatestReactPullRequest.js

```
1 export default function fetchLatestReactPullRequest() {
2   return fetch('https://api.github.com/repos/facebook/react/pulls?state=all&sort=created&direction=desc&per_page=1&page=1', {
3     headers: {
4       'Accept': 'application/vnd.github.v3+json',
5     }
6   })
7   .then(function(response) {
8     if (response.status !== 200) {
9       throw new Error(`Response status code: ${response.status}`);
10    } else {
11      return response.json()
12        .then(function([latestReactPullRequest]) {
13          return {
14            title: latestReactPullRequest.title,
15            body: latestReactPullRequest.body,
16            userLogin: latestReactPullRequest.user.login,
17            createdAt : latestReactPullRequest.created_at,
18          }
19        });
20    }
21  });
22 }
```

#### // src/services/github/actions/fetchLatestReactPullRequest/index.js

```
1 export { default } from './fetchLatestReactPullRequest';
```

#### // src/services/github/index.js

```
1 import fetchLatestReactPullRequest from './actions/fetchLatestReactPullRequest';
2
3 export default {
4   fetchLatestReactPullRequest,
5 };

```

#### // src/components/LatestReactPullRequest/LatestReactPullRequest.js

```
1 // ...
2
3 import githubService from '@services/github';
4 // ...
5
6 componentDidMount() {
7   // ...
8
9   // stage 1
10  const fetchPromise = githubService.fetchLatestReactPullRequest();
11
12  // stage 2
13  // ...
14 }
15
16 // ...

```

By extracting code to the github service, we have freed the component from the task of performing all the micro

operations required to fetch the data. Now it just calls a single method.





## Summary

We started with an example of a React component that had three main tasks and a number of responsibilities behind them.

We extracted the github service from the component and the service can now be reused in other parts of the application.

We unburdened the component from the responsibility of fetching and processing the data from GitHub. The component's code is now a bit simpler.

## #3 Refactoring the data fetching service

The data fetching functionality is now encapsulated within the github service. That's what we wanted to achieve from the perspective of the rest of our application ("client" code).

If fetching the latest pull request from React repo was the only task of the github service, we could just leave it in its current form - even though it has more than one responsibility.

However, in a real-life application it's more likely that we would also fetch other kinds of information from GitHub (for example, a list of open issues) so we would like to reuse some of the functionality.

We left the service in the following form:

// src/services/github/actions/fetchLatestReactPullRequest/fetchLatestReactPullRequest.js

```

1  export default function fetchLatestReactPullRequest() {
2    return fetch('https://api.github.com/repos/facebook/react/pulls?state=all&sort=created&direction=desc&per_page=1&page=1', {
3      headers: {
4        'Accept': 'application/vnd.github.v3+json',
5      }
6    })
7    .then(function(response) {
8      if (response.status !== 200) {
9        throw new Error(`Response status code: ${response.status}`);
10     } else {
11       return response.json()
12     }
13     .then(function([latestReactPullRequest]) {
14       return {
15         title:    latestReactPullRequest.title,
16         body:     latestReactPullRequest.body,
17         userLogin: latestReactPullRequest.user.login,
18         createdAt: latestReactPullRequest.created_at,
19       }
20     });
21   });
22 }
```

We can see that the fetchLatestReactPullRequest function performs two main tasks:

1. Handling request - response operation,
2. Extracting the required data from the response.

Its single responsibility should be the following: "return the result of running the two tasks above, one after the other". Yet, right now it has more responsibilities because it also needs to know the details of how to perform each of these tasks.

Let's extract the request - response handler into fetchData action:

// src/services/github/actions/fetchData/fetchData.js

```

1  export default function fetchData({ path }) {
2    return fetch(`https://api.github.com/${path}`, {
3      headers: {
4        'Accept': 'application/vnd.github.v3+json',
5      }
6    })
7    .then(function(response) {
8      if (response.status !== 200) {
9        throw new Error(`Response status code: ${response.status}`);
10     } else {
11       return response;
12     }
13   })
14   .then(function(response) {
15     return response.json();
16   });
17 }
```

```
// src/services/github/actions/fetchData/index.js
```

```
1 export { default } from './fetchData';
```

We can stop refactoring this piece of code here or break this functionality further.

We perform three steps here:

1. Sending the request,

○ sendRequest:

```
// src/services/github/actions/fetchData/sendRequest.js
```

```
1 export default function sendRequest({ path }) {
2   return fetch(`https://api.github.com/${path}`, {
3     headers: {
4       'Accept': 'application/vnd.github.v3+json',
5     }
6   });
7 }
```

○ throwErrorIfResponseCodeIsDifferentFrom200:

```
// src/services/github/actions/fetchData/throwErrorIfResponseCodeIsDifferentFrom200.js
```

```
1 export default function throwErrorIfResponseCodeIsDifferentFrom200(response) {
2   if (response.status !== 200) {
3     throw new Error(`Response status code: ${response.status}`);
4   } else {
5     return response;
6   }
7 }
```

○ mapResponseToJSON:

```
// src/services/github/actions/fetchData/mapResponseToJSON.js
```

```
1 export default function mapResponseToJSON(response) {
2   return response.json();
3 }
```

And now fetchData becomes:

```
// src/services/github/actions/fetchData/fetchData.js
```

```
1 import sendRequest from './sendRequest';
2 import throwErrorIfResponseCodeIsDifferentFrom200 from './throwErrorIfResponseCodeIsDifferentFrom200';
3 import mapResponseToJSON from './mapResponseToJSON';
4
5 export default function fetchData(requestData) {
6   return sendRequest(requestData)
7     .then(throwErrorIfResponseCodeIsDifferentFrom200)
8     .then(mapResponseToJSON);
9 }
```

Now the responsibility of fetchData is returning the result of the execution of these three steps. It's no longer responsible

2. Converting all non-200 responses to an error,

3. Converting the successful response to JSON.

Let's extract each step to a separate function:

for knowing how each step should be handled - that was just delegated to the newly created functions.

Another benefit of this refactoring is that now the content of `fetchData` reads like **a table of contents we find in books: we only see chapter titles and details are deliberately hidden. If we want to see how an operation is performed, we can go to the specific function in the same way that we navigate from**

**the table of contents to a specific chapter in a book.**

Let's return to `fetchLatestReactPullRequest`, which after extracting `fetchData` becomes:

```
// src/services/github/actions/fetchData/fetchData.js
```

```
1  import fetchData from './fetchData';
2
3  export default function fetchLatestReactPullRequest() {
4    return fetchData({ path: 'repos/facebook/react/pulls?state=all&sort=created&direction=desc&per_page=1&page=1' })
5      .then(function([latestReactPullRequest]) {
6        return {
7          title:    latestReactPullRequest.title,
8          body:     latestReactPullRequest.body,
9          userLogin: latestReactPullRequest.user.login,
10         createdAt : latestReactPullRequest.created_at,
11       }
12     });
13 }
```

Although most of the details about how to communicate with GitHub API went into `fetchData`, we still need to deal with the path of the URL here. That bit couldn't go to `fetchData`

because we want `fetchData` to be reusable for other GitHub API requests. Yet, we know that the path is an extra responsibility which we would like to delegate:

```
// src/services/github/actions/fetchData/fetchData.js
```

```
1  export default function buildFetchDataArgs() {
2    return { path: 'repos/facebook/react/pulls?state=all&sort=created&direction=desc&per_page=1&page=1' };
3  }
```

```
// src/services/github/actions/fetchLatestReactPullRequest/fetchLatestReactPullRequest.js
```

```
1  import buildFetchDataArgs from './buildFetchDataArgs';
2  import fetchData from './fetchData';
3
4  export default function fetchLatestReactPullRequest() {
5    return fetchData(buildFetchDataArgs())
6      .then(function([latestReactPullRequest]) {
7        return {
8          title:    latestReactPullRequest.title,
9          body:     latestReactPullRequest.body,
10         userLogin: latestReactPullRequest.user.login,
11         createdAt : latestReactPullRequest.created_at,
12       }
13     });
14 }
```

Finally, let's take data mapping out:

```
// src/services/github/actions/fetchLatestReactPullRequest/extractDataFromResponseJSON.js
```

```
1  export default function extractDataFromResponseJSON([latestReactPullRequest]) {
2    return {
3      title:    latestReactPullRequest.title,
4      body:     latestReactPullRequest.body,
5      userLogin: latestReactPullRequest.user.login,
6      createdAt : latestReactPullRequest.created_at,
7    }
8  };
```

Similarly to `fetchData`, the single responsibility of `fetchLatestReactPullRequest` is now to glue the subroutines

together. Details are delegated deeper.

```
// src/services/github/actions/fetchLatestReactPullRequest/fetchLatestReactPullRequest.js
```

```
1 import buildFetchDataArgs from './buildFetchDataArgs';
2 import fetchData from './fetchData';
3 import extractDataFromResponseJSON from './extractDataFromResponseJSON';
4
5 export default function fetchLatestReactPullRequest() {
6   return fetchData(buildFetchDataArgs())
7     .then(extractDataFromResponseJSON);
8 }
```

## Allowing to specify the owner and repo

There is still one flaw that inhibits `fetchLatestReactPullRequest` action's code reuse: it has the GitHub's owner and repository names hardcoded. The same is true for the `LatestReactPullRequest` component: it can only show React's latest pull request. That is another responsibility.

These pieces of code are responsible for knowing the owner and repo names.

Let's make the client code tell `LatestReactPullRequest` component what values to use.

Thus far, the component was taking no props:

```
1 <LatestReactPullRequest />
```

Let's add the possibility of specifying the props:

```
1 <LatestGitHubPullRequest owner="facebook" repo="react" />
```

Note that we are also changing the component name here. The old name indicated that the component was coupled with

the React repository. The new name says that the component can handle any specified repository.

```
// src/components/LatestGitHubPullRequest/LatestGitHubPullRequest.js
```

```
1 // ...
2
3 class LatestGitHubPullRequest extends Component {
4   // ...
5
6   componentDidMount() {
7     // ...
8
9     const { owner, repo } = this.props;
10    const fetchPromise = githubService.fetchLatestPullRequest({ owner, repo });
11
12    // ...
13  }
14
15  // ...
16 }
17
18 LatestGitHubPullRequest.propTypes = {
19   owner: PropTypes.string.isRequired,
20   repo: PropTypes.string.isRequired,
21 };
22
23 // ...
```

```
// src/services/github/actions/fetchLatestPullRequest/fetchLatestPullRequest.js
```

```
1 import buildFetchDataArgs from './buildFetchDataArgs';
2 import fetchData from './fetchData';
3 import extractDataFromResponseJSON from './extractDataFromResponseJSON';
4
5 export default function fetchLatestPullRequest({ owner, repo }) {
6   return fetchData(buildFetchDataArgs({ owner, repo }))
7     .then(extractDataFromResponseJSON);
8 }
```

```
src/services/github/actions/fetchLatestPullRequest/buildFetchDataArgs.js
```

```
1 export default function buildFetchDataArgs({ owner, repo }) {
2   return { path: `repos/${owner}/${repo}/pulls?state=all&sort=created&direction=desc&per_page=1&page=1` };
3 }
```

As all the involved pieces of code are no longer tied to the specific React repo, we are also updating names, e.g. latestReactPullRequest to latestPullRequest:

```
// src/services/github/actions/fetchLatestPullRequest/extractDataFromResponseJSON.js
```

```
1 export default function extractDataFromResponseJSON([latestPullRequest]) {
2   return {
3     title: latestPullRequest.title,
4     body: latestPullRequest.body,
5     userLogin: latestPullRequest.user.login,
6     createdAt: latestPullRequest.created_at,
7   }
8 };
```

## Summary

We have refactored the internals of the github service into smaller pieces that can be reused internally when extending the service's functionality.

We have also decoupled the component from the hardcoded owner and repo names, so that it can handle any given GitHub repository.

One of the most important gains that we achieved here is **reusability**. These pieces of code are now more open to changes and adding new functionality.

We are now leaving the github service. In the next chapter, we will return to the LatestGitHubPullRequest component to see how we can delegate its other responsibilities.

## #4 Freeing the component from data fetching

We left the LatestGitHubPullRequest component in the following state:

// src/components/LatestGitHubPullRequest/LatestGitHubPullRequest.js

```

1  import React, { Component, Fragment } from 'react';
2  import PropTypes from 'prop-types';
3  import moment from 'moment';
4  import makePromiseCancelable from '@utils/makePromiseCancelable';
5  import githubService from '@services/github';
6  import './LatestGitHubPullRequest.css';
7
8  const DATA_FETCHING_STATUS = {
9    NOT_STARTED: Symbol('DATA_FETCHING_STATUS_NOT_STARTED'),
10   IN_PROGRESS: Symbol('DATA_FETCHING_STATUS_IN_PROGRESS'),
11   SUCCESS:     Symbol('DATA_FETCHING_STATUS_SUCCESS'),
12   FAILURE:     Symbol('DATA_FETCHING_STATUS_FAILURE'),
13 };
14
15 class LatestGitHubPullRequest extends Component {
16   constructor(props) {
17     super(props);
18     this.state = {
19       dataFetchingStatus: DATA_FETCHING_STATUS.NOT_STARTED,
20       data: null,
21     };
22   }
23
24   componentDidMount() {
25     this.setState({
26       dataFetchingStatus: DATA_FETCHING_STATUS.IN_PROGRESS,
27     });
28
29     const { owner, repo } = this.props;
30
31     const fetchPromise = githubService.fetchLatestPullRequest({ owner, repo });
32
33     const { promise, cancel } = makePromiseCancelable(fetchPromise);
34     this.cancelDataFetchingPromise = cancel;
35
36     promise
37       .then((data) => {
38         this.setState({
39           dataFetchingStatus: DATA_FETCHING_STATUS.SUCCESS,
40           data
41         });
42       })
43       .catch((error) => {
44         if (error.isCanceled) return;
45
46         this.setState({
47           dataFetchingStatus: DATA_FETCHING_STATUS.FAILURE,
48           data: null,
49         });
50       });
51   }
52
53   componentWillUnmount() {
54     this.cancelDataFetchingPromise();
55   }
56
57   render() {
58     const { dataFetchingStatus, data } = this.state;
59     return (
60       <div className="c-latest-github-pull-request">
61         {dataFetchingStatus === DATA_FETCHING_STATUS.NOT_STARTED && (
62           <Fragment>Initializing...</Fragment>)
63       }

```

```

64 {dataFetchingStatus === DATA_FETCHING_STATUS.IN_PROGRESS && (
65   <Fragment>Fetching...</Fragment>
66 )}
67 {dataFetchingStatus === DATA_FETCHING_STATUS.FAILURE && (
68   <Fragment>Data fetching error...</Fragment>
69 )}
70 {dataFetchingStatus === DATA_FETCHING_STATUS.SUCCESS && (
71   <Fragment>
72     <div className="c-latest-github-pull-request__title">{data.title}</div>
73     <div className="c-latest-github-pull-request__body">{data.body}</div>
74     <div className="c-latest-github-pull-request__created-at-and-user-login">
75       {moment(data.createdAt).calendar()} by {data.userLogin}
76     </div>
77   </Fragment>
78 )}
79 </div>
80 );
81 }
82 }
83
84 LatestGitHubPullRequest.propTypes = {
85   owner: PropTypes.string.isRequired,
86   repo:  PropTypes.string.isRequired,
87 };
88
89 export default LatestGitHubPullRequest;

```

Although the component doesn't need to know from where and how to handle data fetching, it still supervises the process. It calls the `githubService.fetchLatestPullRequest(...)` method, tracks the fetching status, and cancels the fetch promise if the component happens to be unmounted before the promise resolves.

Note that the mentioned functionality belongs to a **logically separate layer**. It's responsible for managing the **data fetching process**, while the rest of the component is responsible for **displaying information**.





Let's reflect this in code. What if we extract the data fetching management code to a wrapper component which would then be passing the data and the status down to

LatestGitHubPullRequest in props? A higher-order component (hoc) looks like a good candidate for the job:

// src/hoc/withLatestGitHubPullRequest.js

```
1  import React, { Component } from 'react';
2  import PropTypes from 'prop-types';
3  import getComponentDisplayName from '@hoc/utils/getComponentDisplayName';
4  import makePromiseCancelable from '@utils/makePromiseCancelable';
5  import githubService from '@services/github';
6  import DATA_FETCHING_STATUS from '@consts/dataFetchingStatus';
7
8  export default function withLatestGitHubPullRequest(OriginalComponent) {
9    class WithLatestGitHubPullRequest extends Component {
10      constructor(props) {
11        super(props);
12        this.state = {
13          dataFetchingStatus: DATA_FETCHING_STATUS.NOT_STARTED,
14          data: null,
15        };
16      }
17
18      componentDidMount() {
19        this.setState({
20          dataFetchingStatus: DATA_FETCHING_STATUS.IN_PROGRESS,
21        });
22
23        const { owner, repo } = this.props;
24        const fetchPromise = githubService.fetchLatestPullRequest({ owner, repo });
25        const { promise, cancel } = makePromiseCancelable(fetchPromise);
26
27        this.cancelDataFetchingPromise = cancel;
28
29        promise
30          .then((data) => {
31            this.setState({
32              dataFetchingStatus: DATA_FETCHING_STATUS.SUCCESS,
33              data
34            });
35          })
36          .catch((error) => {
37            if (error.isCanceled) return;
38
39            this.setState({
40              dataFetchingStatus: DATA_FETCHING_STATUS.FAILURE,
41              data: null,
42            });
43          });
44      }
45
46      componentWillUnmount() {
47        this.cancelDataFetchingPromise();
48      }
49
50      render() {
51        const { owner, repo, ...otherProps } = this.props;
52        const { dataFetchingStatus, data } = this.state;
53        return (
54          <OriginalComponent
55            {...otherProps}
56            dataFetchingStatus={dataFetchingStatus}
57            data={data}
58          />
59        );
60      }
61    }
62  }
```

```

63 WithLatestGitHubPullRequest.displayName = `WithLatestGitHubPullRequest(${getComponentDisplayName(OriginalComponent)})`;
64
65 WithLatestGitHubPullRequest.propTypes = {
66   owner: PropTypes.string.isRequired,
67   repo: PropTypes.string.isRequired,
68 };
69
70 return WithLatestGitHubPullRequest;
71 }

```

The original component gets thinner:

// src/components/GitHubPullRequest/GitHubPullRequest.js

```

1  import React, { Fragment } from 'react';
2  import PropTypes from 'prop-types';
3  import moment from 'moment';
4  import DATA_FETCHING_STATUS from '@consts/dataFetchingStatus';
5  import './GitHubPullRequest.css';
6
7  const GitHubPullRequest = ({ dataFetchingStatus, data }) => (
8    <div className="c-latest-github-pull-request">
9      {dataFetchingStatus === DATA_FETCHING_STATUS.NOT_STARTED && (
10        <Fragment>Initializing...</Fragment>)
11      }
12      {dataFetchingStatus === DATA_FETCHING_STATUS.IN_PROGRESS && (
13        <Fragment>Fetching...</Fragment>
14      )}
15      {dataFetchingStatus === DATA_FETCHING_STATUS.FAILURE && (
16        <Fragment>Data fetching error...</Fragment>
17      )}
18      {dataFetchingStatus === DATA_FETCHING_STATUS.SUCCESS && (
19        <Fragment>
20          <div className="c-latest-github-pull-request__title">{data.title}</div>
21          <div className="c-latest-github-pull-request__body">{data.body}</div>
22          <div className="c-latest-github-pull-request__created-at-and-user-login">
23            {moment(data.createdAt).calendar()} by {data.userLogin}
24          </div>
25        </Fragment>
26      )}
27    </div>
28  );
29
30 GitHubPullRequest.propTypes = {
31   dataFetchingStatus: PropTypes.oneOf(
32     Object.values(DATA_FETCHING_STATUS)
33   ).isRequired,
34   data: PropTypes.shape({
35     title: PropTypes.string.isRequired,
36     body: PropTypes.string.isRequired,
37     userLogin: PropTypes.string.isRequired,
38     createdAt: PropTypes.string.isRequired,
39   }),
40 };
41
42 export default GitHubPullRequest;

```

See that we also removed the word Latest from the component's name, as now it's **no longer coupled specifically with the latest pull request**. It can now display any pull

request data provided via props. **The component's reusability was improved.**

We moved DATA\_FETCHING\_STATUS to a separate file, as it's now used in more than one place:

// src/consts/dataFetchingStatus.js

```
1 export default {
2   NOT_STARTED: Symbol('DATA_FETCHING_STATUS_NOT_STARTED'),
3   IN_PROGRESS: Symbol('DATA_FETCHING_STATUS_IN_PROGRESS'),
4   SUCCESS: Symbol('DATA_FETCHING_STATUS_SUCCESS'),
5   FAILURE: Symbol('DATA_FETCHING_STATUS_FAILURE'),
6 };
```

And the original component is now wrapped with:

// src/containers/LatestGitHubPullRequest/LatestGitHubPullRequest.js

```
1 import GitHubPullRequest from '@components/GitHubPullRequest';
2 import withLatestGitHubPullRequest from '@hoc/withLatestGitHubPullRequest';
3
4 export default withLatestGitHubPullRequest(GitHubPullRequest);
```

Let's focus for a moment on the extracted withLatestGitHubPullRequest hoc. If it's the only component in the application that fetches data and tracks the fetching status, we could leave it in its current shape.

But let's say we have another component that fetches some other data. It can be a list of open React issues on GitHub, or any other information from any other service in the internet. Then both of these two components would probably contain the same status tracking code.

**We see that withLatestGitHubPullRequest fetching status tracking is coupled to a specific data fetch**, namely: githubService.fetchLatestPullRequest(...). The other component would track the status in the same way. The only difference would be the function that it calls to get the data.

**Perhaps we could pass the data fetching function as an argument, thereby making the status tracking code independent from the actual data source?**

// src/hoc/withDataFetching.js

```
1 import React, { Component } from 'react';
2 import PropTypes from 'prop-types';
3 import getComponentDisplayName from '@hoc/utils/getComponentDisplayName';
4 import makePromiseCancelable from '@utils/makePromiseCancelable';
5 import DATA_FETCHING_STATUS from '@consts/dataFetchingStatus';
6
7 export default function withDataFetching(OriginalComponent) {
8   class WithDataFetching extends Component {
9     constructor(props) {
10       super(props);
11       this.state = {
12         dataFetchingStatus: DATA_FETCHING_STATUS.NOT_STARTED,
13         data: null,
14       };
15     }
16
17     componentDidMount() {
18       this.setState({
19         dataFetchingStatus: DATA_FETCHING_STATUS.IN_PROGRESS,
20       });
21
22       const { fetchData } = this.props;
23       const { promise, cancel } = makePromiseCancelable(fetchData());
24       this.cancelDataFetchingPromise = cancel;
25
26       promise
27         .then((data) => {
28           this.setState({
29             dataFetchingStatus: DATA_FETCHING_STATUS.SUCCESS,
30             data
31           });
32         })
33         .catch(() => {
34           this.setState({
35             dataFetchingStatus: DATA_FETCHING_STATUS.FAILURE,
36             data: null
37           });
38         });
39     }
40
41     render() {
42       return <OriginalComponent {...this.props} />;
43     }
44   }
45
46   WithDataFetching.propTypes = {
47     fetchData: PropTypes.func.isRequired,
48   };
49
50   return WithDataFetching;
51 }
```

```

32     })
33     .catch((error) => {
34         if (error.isCanceled) return;
35
36         this.setState({
37             dataFetchingStatus: DATA_FETCHING_STATUS.FAILURE,
38             data: null,
39         });
40     });
41 }
42
43 componentWillUnmount() {
44     this.cancelDataFetchingPromise();
45 }
46
47 render() {
48     const { fetchData, ...otherProps } = this.props;
49     const { dataFetchingStatus, data } = this.state;
50     return (
51         <OriginalComponent
52             {...otherProps}
53             dataFetchingStatus={dataFetchingStatus}
54             data={data}
55         />
56     );
57 }
58 }
59
60 WithDataFetching.displayName = `WithDataFetching(${getComponentDisplayName(OriginalComponent)})`;
61
62 WithDataFetching.propTypes = {
63     fetchData: PropTypes.func.isRequired,
64 };
65
66 return WithDataFetching;
67 }

```

We extracted most of the code into `withDataFetching` hoc. It takes `fetchData` function via props.

The contract is that the function returns a promise. And that's it. The `withDataFetching` doesn't know what data are fetched and from where. The only responsibility it has is tracking

when the fetching starts and whether it succeeds or fails.

**That means `withDataFetching` hoc is now reusable as it can supervise fetching from any data source.**



Our `withLatestGitHubPullRequest` hoc is now responsible for building the `fetchData` function and passing it to the data fetching component.

// `src/hoc/withLatestGitHubPullRequest.js`

```
1  import React from 'react';
2  import PropTypes from 'prop-types';
3  import getComponentDisplayName from '@hoc/utils/getComponentDisplayName';
4  import githubService from '@services/github';
5
6  export default function withLatestGitHubPullRequest(OriginalComponent) {
7    const WithLatestGitHubPullRequest = function({ owner, repo, ...otherProps }) {
8      const fetchData = function() {
9        return githubService.fetchLatestPullRequest({ owner, repo });
10     };
11     return (
12       <OriginalComponent
13         {...otherProps}
14         fetchData={fetchData}
15       />
16     );
17   }
18
19   WithLatestGitHubPullRequest.displayName = `WithLatestGitHubPullRequest(${getComponentDisplayName(OriginalComponent)})`;
20
21   WithLatestGitHubPullRequest.propTypes = {
22     owner: PropTypes.string.isRequired,
23     repo:  PropTypes.string.isRequired,
24   };
25
26   return WithLatestGitHubPullRequest;
27 }
```

And now we put it all together:

// `src/containers/LatestGitHubPullRequest/LatestGitHubPullRequest.js`

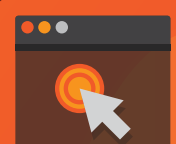
```
1  import GitHubPullRequest from '@components/GitHubPullRequest';
2  import withDataFetching from '@hoc/withDataFetching';
3  import withLatestGitHubPullRequest from '@hoc/withLatestGitHubPullRequest';
4
5  export default withLatestGitHubPullRequest(withDataFetching(GitHubPullRequest));
6
```

Finally, let's return to the `GitHubPullRequest` component.

It now has two responsibilities:

1. displaying the fetching status, and
2. displaying the pull request information.

Let's relieve the component from the first responsibility.



We can do that by extracting the code to DataFetchingStatus component. Its only responsibility is displaying the status information:

// src/components/DataFetchingStatus/DataFetchingStatus.js

```
1 import React, { Fragment } from 'react';
2 import PropTypes from 'prop-types';
3 import DATA_FETCHING_STATUS from '@consts/dataFetchingStatus';
4 import './DataFetchingStatus.css';
5
6 const DataFetchingStatus = ({ dataFetchingStatus, data }) => (
7   <div className="c-data-fetching-status">
8     {dataFetchingStatus === DATA_FETCHING_STATUS.NOT_STARTED && (
9       <Fragment>Initializing...</Fragment>)
10    }
11    {dataFetchingStatus === DATA_FETCHING_STATUS.IN_PROGRESS && (
12      <Fragment>Fetching...</Fragment>
13    )}
14    {dataFetchingStatus === DATA_FETCHING_STATUS.FAILURE && (
15      <Fragment>Data fetching error...</Fragment>
16    )}
17    {dataFetchingStatus === DATA_FETCHING_STATUS.SUCCESS && (
18      <Fragment>Data fetched successfully.</Fragment>
19    )}
20   </div>
21 );
22
23 DataFetchingStatus.propTypes = {
24   dataFetchingStatus: PropTypes.oneOf(
25     Object.values(DATA_FETCHING_STATUS)
26   ).isRequired,
27 };
28
29 export default DataFetchingStatus;
```

And GitHubPullRequest is now left with the responsibility to display the pull request information:

// src/components/GitHubPullRequest/GitHubPullRequest.js

```
1 import React from 'react';
2 import PropTypes from 'prop-types';
3 import moment from 'moment';
4 import './GitHubPullRequest.css';
5
6 const GitHubPullRequest = ({ data: { title, body, userLogin, createdAt } }) => (
7   <div className="c-latest-github-pull-request">
8     <div className="c-latest-github-pull-request__title">{title}</div>
9     <div className="c-latest-github-pull-request__body">{body}</div>
10    <div className="c-latest-github-pull-request__created-at-and-user-login">
11      {moment(createdAt).calendar()} by {userLogin}
12    </div>
13   </div>
14 );
15
16 GitHubPullRequest.propTypes = {
17   data: PropTypes.shape({
18     title: PropTypes.string.isRequired,
19     body: PropTypes.string.isRequired,
20     userLogin: PropTypes.string.isRequired,
21     createdAt: PropTypes.string.isRequired,
22   }),
23 };
24
25 export default GitHubPullRequest;
```

We could extract the creation date formatting functionality even further, but for the purpose of this article we will stop here.

How do we make DataFetchingStatus and GitHubPullRequest work together? Let's use another hoc.

#### // src/hoc/withDataFetchingStatus.js

```
1 import React from 'react';
2 import PropTypes from 'prop-types';
3 import getComponentDisplayName from '@hoc/utils/getComponentDisplayName';
4 import DATA_FETCHING_STATUS from '@consts/dataFetchingStatus';
5
6 export default function withDataFetchingStatus(DataFetchingStatus) {
7   return function(OriginalComponent) {
8     const WithDataFetchingStatus = function({ dataFetchingStatus, ...otherProps }) {
9       return (
10         (dataFetchingStatus === DATA_FETCHING_STATUS.SUCCESS
11           ? <OriginalComponent {...otherProps} />
12           : <DataFetchingStatus dataFetchingStatus={dataFetchingStatus} />
13         )
14       );
15     };
16
17     WithDataFetchingStatus.displayName = `WithDataFetchingStatus(${getComponentDisplayName(OriginalComponent)})`;
18
19     WithDataFetchingStatus.propTypes = {
20       dataFetchingStatus: PropTypes.oneOf(
21         Object.values(DATA_FETCHING_STATUS)
22       ).isRequired,
23     };
24
25     return WithDataFetchingStatus;
26   };
27 }
```

#### // src/hoc/withDataFetchingStatus.js

```
1 import GitHubPullRequest from '@components/GitHubPullRequest';
2 import DataFetchingStatus from '@components/DataFetchingStatus';
3 import withDataFetching from '@hoc/withDataFetching';
4 import withDataFetchingStatus from '@hoc/withDataFetchingStatus';
5 import withLatestGitHubPullRequest from '@hoc/withLatestGitHubPullRequest';
6
7 export default withLatestGitHubPullRequest(
8   withDataFetching(
9     withDataFetchingStatus(DataFetchingStatus)(
10       GitHubPullRequest
11     )
12   )
13 );
```

In this particular setup, the DataFetchingStatus won't have opportunity to render the information about success, because itself it's not rendered in such scenario; the actual data is.

Yet we've given DataFetchingStatus this functionality to make it a fully autonomous component, handling all the statuses.



## Summary

In this chapter, we split the React component that we started with in Part 1 further. We extracted separate parts responsible for:

- Deciding which data fetching functionality to use (withLatestGitHubPullRequest hoc),
- Managing the fetching process (withDataFetching hoc),
- Displaying the fetching status (DataFetchingStatus component),
- Displaying the pull request data (GitHubPullRequest component),
- Deciding whether to display the status or the data (withDataFetchingStatus hoc).

Along with previously extracted github service, these are now building blocks that comprise the functionality to display the latest React repo pull request.

Note that the total number of code lines is higher now than it was at the beginning. The main reasons behind that are the extra code that we now need to glue the parts together and the language syntax overhead.

**Yet, the code blocks themselves are smaller and simpler. There are fewer execution paths inside each of them. And it's usually easier to understand the system when analysing it one part at a time rather than everything at once.**

Once the original fat React component was decomposed into parts, these parts could now be reused to extend the existing functionality with a minimal amount of new code required.

Having single-responsibility building blocks at our disposal, we can now easily select those that offer what is necessary for a particular new feature; none of them carry the baggage of unwanted functionality.



## Conclusion

The three chapters showed us that React components don't need to be fat. We just have to be aware of the responsibilities each piece of code has and then consciously decide whether we want to delegate them.

 *Note: The Single Responsibility Principle can be applied to brand-new React components as well as to the existing fat ones through refactoring.*

**Encapsulating each responsibility in a separate function, class, or service gives us valuable separation of concerns. It's far easier to find a piece where a specific functionality is implemented, isolate buggy code, or simply replace an existing element with a new one.**

The code is much more testable as well. A smaller number of code execution paths means there are fewer scenarios to test. Each part can be unit-tested separately so that we know our building blocks work properly.

All this makes the software more reliable. Any code changes and extensions are cheaper and quicker.

**Got a project on your mind?**

[Hire us](#)

**Looking for a new challenge?**

[Join our team](#)



Sunsrapers is a software development company made of Python and JavaScript experts skilled in web & mobile development, and data science.

We partner with startups, SMBs, and enterprises to help them solve problems, achieve business goals, and automate processes using technology. We combine business-savvy software craftsmanship and agile project management to build state-of-the-art applications that follow industry standards.

We delivered 150+ digital products to clients all over the world, including: ASUS, Caris Life Sciences, Samsung, Lexmark, Logitech, Open Finance, PKO, Ricoh, as well as many startups.

Are you looking for a trusted tech partner? Drop us a line - we'd love to hear from you and see how we can help realize your vision.



**New Business**

[hello@sunsrapers.com](mailto:hello@sunsrapers.com)

**Careers**

[careers@sunsrapers.com](mailto:careers@sunsrapers.com)



**Find us**

Sunsrapers Sp. z o.o.  
Pokorna 2/947  
00-199 Warsaw, Poland  
VAT EU: PL1182116268

