Manujith Pallewatte

Optimizing Data Fetching in React

Manujith Pallewatte
Jun 5, 2020 • 15 Min read • 3,709 Views

Jun 5, 2020 • 15 Min read • 3,709 Views

Web Development Front End Web Dev... Client-side Framew... React

Introduction

12

- Introduction
- The Auto-Complete Search
- Debouncing
- Throttling
- Request-Response Matching
- Conclusion
- Top

Introduction

Data fetching is a core requirement of almost every real-world React app. Sometimes we are required to interact with APIs that are being called in rapid succession from our web app. An auto-complete search is one example. If these instances are not carefully planned, we could easily end up degrading the performance of the app as well as the API. In this guide, we explore several techniques to optimize data fetch requests and the effect of them.

To demonstrate the ideas discussed in this guide, we will use a simple auto-complete search. Auto-complete is a function expected in any modern app, yet it is often a poorly engineered feature due to its simplicity. Most of the time, developers put thought into providing the best visual output of the component, but the actual issues of data fetching are lost.

The Auto-Complete Search

First, initialize a React-Redux project with one `search` action added, which will be used to retrieve the search results for the keywords. To keep the guide focus intact, only certain components of the app will be discussed here. But you can find the complete source code at this Github Repo. To provide the auto-complete UI, you can install the react-autocomplete library from npm. With all the components added and initialized, `App.js` will look as follows.

jsx

```jsx
import React, { useState } from 'react';
import './App.css';
import { Provider, useSelector, useDispatch } from 'react-redux'
import store from './store';
import Autocomplete from 'react-autocomplete';
import { search } from './actions';

```

```
function SearchComponent(){
    const [query, setQuery] = useState("");
    const hits = useSelector(state => state.hits);
    const results = useSelector(state => state.searchResults);

    const dispatch = useDispatch();

    const onSearch = (query) => {
        setQuery(query);
        dispatch(search(query));
    }

    return (
        <div>
            <div>API hits: {hits}</div>
            <Autocomplete
                getItemValue={(item) => item}
                items={results}
                renderItem={(item, isHighlighted) =>
                    <div style={{ background: isHighlighted ? 'lightgray'
                        {item}
                    </div>
                }
                value={query}
                onChange={(e) => onSearch(e.target.value)}
            />
        </div>
    )
}

function App() {
    return (
        <Provider store={store}>
            <div className="App">
                <h2>Auto-complete Search</h2>
                <SearchComponent />
            </div>
        </Provider>
    )
}

export default App;
```

Note that you access two variables from the Redux store.

1. `searchResults`: matching strings for a given query
2. `hits`: number of times the API is called

`hits` will be used to demonstrate the importance of considering the query optimizations and the effect of each optimization technique. Before you run the above code, take a look at `mockapi.js`, where a search function is created to mimic the actions of a search API.

```js
import mockData from './mockdata.json';

var hits = 0;

const sleep = (milliseconds) => {
    return new Promise(resolve => setTimeout(resolve, milliseconds))
}

export async function searchAPI(query){
    await sleep(50);
```

```
        return {
            data: {
                results: mockData.filter(item => item.toLowerCase().includes(
                hits: ++hits
            }
        }
    }
```

A simple counter is used to track the number of API hits to this mock search API. Now run the complete code and try typing in few keywords, such as *samsung* and *samsung galaxy*. You will notice that while you are typing, with each keystroke the number of hits increases, meaning that your API has been called that many times. Most of the intermediate API calls were unnecessary. So today's goal is to reduce the number of hits without degrading the user experience.

Debouncing

Debouncing is a form of action delay where a defined period is observed after the last call to a function is fired. This means that if a user is typing a word, the app buffers all search calls until the user stops typing, and then waits for another period to see if the user starts typing again. If not, then it fires the last received function call. Considering the above autocomplete function, for instance, this means that for the search query *samsung s10* it would fire only one `search` action with the query parameter as *samsung s10*. Earlier, it would have dispatched ten actions, one for every keystroke of the user.

Apart from saving the bandwidth of your API, debouncing also prevents excessive re-rendering of React components. For example, if you are showing the results of the query live while typing, for each response received, the result component would be re-rendered.

There are several ways to implement debouncing in React. You could either write your own debounce implementation, or use a library that would provide the functionality. Some widely used libraries are:

1. lodash
2. underscore
3. RxJS

In this guide, you will use `lodash`. If you would rather implement the functionality yourself, this article provides excellent examples. You can install `lodash` with `npm install --save lodash`.

The following example shows how the `lodash` debounce function is used to limit the action calls.

javascript

```javascript
// actions.js

import * as mockAPI from './mockapi';
import debounce from 'lodash/debounce';

// These are our action types
export const SEARCH_REQUEST = "SEARCH_REQUEST"
export const SEARCH_SUCCESS = "SEARCH_SUCCESS"
export const SEARCH_ERROR = "SEARCH_ERROR"


// Now we define actions
export function searchRequest(){
    return {
        type: SEARCH_REQUEST
    }
}

export function searchSuccess(payload){
    return {
        type: SEARCH_SUCCESS,
        payload
    }
}

export function searchError(error){
    return {
        type: SEARCH_ERROR,
        error
    }
}

export function search(query) {
    return async function (dispatch) {
        dispatch(searchRequest());
        try{
            const response = await mockAPI.searchAPI(query);
            dispatch(searchSuccess(response.data));
        }catch(error){
            dispatch(searchError(error));
        }
    }
}

// debouncing the searchAPI method with wait time of 800 miliseconds
// note that we have leading=true, means that initial function call will
// before starting the wait time
// withut it return value is initial null, which will break the search cc
const debouncedSearchAPI = debounce(async (query) => {
    return await mockAPI.searchAPI(query)
}, 800, { leading: true });

export function debouncedSearch(query) {
    return async function (dispatch) {
        dispatch(searchRequest());
        try{
            const response = await debouncedSearchAPI(query);
            dispatch(searchSuccess(response.data));
        }catch(error){
            dispatch(searchError(error));
        }
    }
}
```

`SearchPage` needs to be modified to use the debounced search in place of the regular search as well.

jsx

```jsx
// SearchPage.js
// ....

    const onSearch = (query) => {
            setQuery(query);
            dispatch(debouncedSearch(query));
    }

// ...
```

Now, try running the same set of queries as above and observe the change in the number of API hits. You can experiment with the buffer time in the debounce function to get a better idea of the effect of debouncing. While this prevents the problem of rapid API call execution, now the user experience seems to be degraded. An ideal auto-complete should provide suggestions while the user is typing, whereas the current implementation would only provide suggestions after the user has stopped typing. To correct this, you could use the next optimization technique: throttling.

Throttling

Throttling provides optimization by limiting the number of actions of the same function being called in a given interval of time. For example, you could determine that the search action should only be fired every 100 milliseconds. This way, regardless of whether the user is still typing or not, the `search` action will be dispatched once every 100 milliseconds, improving the user experience.

You can use the same library you used for debouncing for throttling as well. The following example has modified the previous code to implement throttling.

javascript

```javascript
// actions.js

import * as mockAPI from './mockapi';
import debounce from 'lodash/debounce';
import throttle from 'lodash/throttle';

// These are our action types
export const SEARCH_REQUEST = "SEARCH_REQUEST"
export const SEARCH_SUCCESS = "SEARCH_SUCCESS"
export const SEARCH_ERROR = "SEARCH_ERROR"


// Now we define actions
export function searchRequest(){
    return {
        type: SEARCH_REQUEST
    }
}

```

```
export function searchSuccess(payload){
    return {
        type: SEARCH_SUCCESS,
        payload
    }
}

export function searchError(error){
    return {
        type: SEARCH_ERROR,
        error
    }
}

export function search(query) {
    return async function (dispatch) {
        dispatch(searchRequest());
        try{
            const response = await mockAPI.searchAPI(query);
            dispatch(searchSuccess(response.data));
        }catch(error){
            dispatch(searchError(error));
        }
    }
}

const debouncedSearchAPI = debounce(async (query) => {
    return await mockAPI.searchAPI(query)
}, 800, { leading: true });

export function debouncedSearch(query) {
    return async function (dispatch) {
        dispatch(searchRequest());
        try{
            const response = await debouncedSearchAPI(query);
```

```
                dispatch(searchSuccess(response.data));
            }catch(error){
                dispatch(searchError(error));
            }
        }
    }

    const throttledSearchAPI = throttle(async (query) => {
        return await mockAPI.searchAPI(query)
    }, 300, { leading: true });

    export function throttledSearch(query) {
        return async function (dispatch) {
            dispatch(searchRequest());
            try{
                const response = await throttledSearchAPI(query);
                dispatch(searchSuccess(response.data));
            }catch(error){
                dispatch(searchError(error));
            }
        }
    }
```

jsx

```
// SearchPage.js
// ....

    const onSearch = (query) => {
            setQuery(query);
            dispatch(throttledSearch(query));
    }

// ...
```

Make the changes and observe how the search experience is now changed with throttling when compared to debouncing alone. Note that each technique has its use cases. So it's the developer's task to select the correct tool.

Request-Response Matching

By using either of the above techniques, you can significantly improve the data-fetching aspects of your web apps. Yet, when it comes to a real-world application, there are unforeseen complications, such as network delays and unpredictable processing times in database servers. As a result, your API might not return results in the same order that you sent them.

Assume that in the above scenario, two consecutive search queries are dispatched to a real API as follows.

1. `/search?q=samsung galaxy`
2. `/search?q=samsung galaxy watch`

This would yield two responses. Response 1:

```
{
    "data": {
        "results": ["samsung galaxy", "samsung galaxy s10", ...]
        "hits": <hit number>
    }
}
```

Response 2:

```
{
    "data": {
        "results": ["samsung galaxy watch", "samsung galaxy watch 2", ...]
        "hits": <hit number>
    }
}
```

It is apparent that the user's last query here is `samsung galaxy watch`. So, the auto-complete should ideally show results that include `samsung galaxy watch` rather than just `samsung galaxy`. But due to network delays, Response 1 may arrive after Response 2. This means the auto-complete is showing an incorrect result set to the user.

To prevent such occurrences, you can use a request-response mapping technique. This generally involves making changes to your API as well as to your web app. For the above scenario, you could resolve by changing the API to return the search query along with the search results. Then the response can be validated by matching the returned query with the stored query.

js

```js
// mockapi.js

export async function searchAPI(query){
    await sleep(50);
    return {
        data: {
            results: mockData.filter(item => item.toLowerCase().includes(
            hits: ++hits,
            query: query
        }
    }
}
```

Reducer is updated to store the current user query before dispatching the API call. The `searchRequest` action reducer facilitates this. And on receiving the search results, the query returned from the API is compared to determine if the results should be updated or not.

js

```js
// reducers.js

import { SEARCH_REQUEST, SEARCH_SUCCESS } from './actions';

const initialState = {
    searchResults: [],
    hits: 0,
    currentQuery: ""
}

export default function searchReducer(state, action) {
    if (typeof state === 'undefined') {
        return initialState
    }

    switch(action.type){
        case SEARCH_REQUEST:
            state.currentQuery = action.payload;
            break;

        case SEARCH_SUCCESS:
            if(state.currentQuery === action.payload.query){
                state.searchResults = action.payload.results;
                state.hits = action.payload.hits;
            }
            break;
    }

    return state
}
```

js

```js
// actions.js

// ...
export function search(query) {
    return async function (dispatch) {
        dispatch(searchRequest(query));
        try{
            const response = await mockAPI.searchAPI(query);
```

```
                dispatch(searchSuccess(response.data));
            }catch(error){
                dispatch(searchError(error));
            }
        }
    }
    //...
```

Note that there is no one correct solution for integrating such mappings. The solution will depend on your API architecture and the web app. But as a practice, tagging a uniquely generated ID with a request can be used.

Conclusion

In this guide, we explored a simple use case of an auto-complete search and its effect on API load if optimization is not done in data fetching. We used debouncing and throttling as two techniques to optimize it and observed how each would affect the API load as well as the user experience. Finally, we improved the solution further by request-response mapping to consider practical errors that can occur in a real-world application.