

Apollo Client Optimistic UI in React

This lesson introduces you to Optimistic UI in general and how to implement it using Apollo Client in React.

WE'LL COVER THE FOLLOWING ^

- What is Optimistic UI?
- Introducing `optimisticResponse`
- Exercises
- Reading Tasks

We've covered the basics, so now it's time to move to the advanced topics. One of those topics is the **optimistic UI** with React Apollo, which makes everything onscreen more synchronous.

What is Optimistic UI?

For instance, when liking a post on Twitter, the like appears immediately. As developers, we know there is a request that sends that *"like"* information to the Twitter backend. This request is asynchronous and does not resolve immediately with a result. The optimistic UI immediately assumes a successful request and mimics the result of such request for the frontend so it can update its UI immediately before the real response arrives later. With a failed request, the optimistic UI performs a rollback and updates itself accordingly. Optimistic UI improves the user experience by omitting inconvenient feedback (e.g. loading indicators) for the user. The good thing is that the React Apollo comes with this feature out of the box.

In this lesson, we will implement an optimistic UI for the situation when a user clicks the watch/unwatch mutation we have implemented.

First, let's have a look at the mutation:

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
const WATCH_REPOSITORY = gql`
  mutation ($id: ID!, $viewerSubscription: SubscriptionState!) {
    updateSubscription(
      input: { state: $viewerSubscription, subscribableId: $id }
    ) {
      subscribable {
        id
        viewerSubscription
      }
    }
  }
`;
```

src/Repository/mutation.js

Secondly, here goes the usage of the mutation with a **Mutation** render prop component:

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
const VIEWER_SUBSCRIPTIONS = {
  SUBSCRIBED: 'SUBSCRIBED',
  UNSUBSCRIBED: 'UNSUBSCRIBED',
};

const isWatch = viewerSubscription =>
  viewerSubscription === VIEWER_SUBSCRIPTIONS.SUBSCRIBED;

const updateWatch = () => {
  ...
};

const RepositoryItem = ({ ... }) => (
  <div>
    <div className="RepositoryItem-title">
      ...

    <div>
      ...

      <Mutation
        mutation={WATCH_REPOSITORY}
        variables={{
          id,
```

```

        viewerSubscription: isWatch(viewerSubscription)
        ? VIEWER_SUBSCRIPTIONS.UNSUBSCRIBED
        : VIEWER_SUBSCRIPTIONS.SUBSCRIBED,

      })
      update={updateWatch}
    >
    {(updateSubscription, { data, loading, error }) => (
      <Button
        className="RepositoryItem-title-action"
        onClick={updateSubscription}
      >
        {watchers.totalCount}{' '}
        {isWatch(viewerSubscription) ? 'Unwatch' : 'Watch'}
      </Button>
    )}
  </Mutation>

  ...
</div>
</div>

...
</div>
);

```

src/Repository/RepositoryItem/index.js

And thirdly, here is the `updateWatch` function that is passed to the `Mutation` component:

Environment Variables



Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```

const updateWatch = (
  client,
  {
    data: {
      updateSubscription: {
        subscribable: { id, viewerSubscription },
      },
    },
  },
) => {
  const repository = client.readFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
  });

  let { totalCount } = repository.watchers;
  totalCount =
    viewerSubscription === VIEWER_SUBSCRIPTIONS.SUBSCRIBED
      ? totalCount + 1
      : totalCount - 1;

```



```

    totalCount,
  },
  repository: repository,
  watchers: {
    repository: repository,
    totalCount,
  },
},
});
};

```

src/Repository/RepositoryItem/index.js

Introducing **optimisticResponse**

Now let's get to the optimistic UI feature. Fortunately, the **Mutation** component offers a prop for the optimistic UI strategy called **optimisticResponse**. It returns the same result, which is accessed as an argument in the function passed to the **update** prop of the **Mutation** component. With a watch mutation, only the **viewerSubscription** status changes to **subscribed** or **unsubscribed**. This is an optimistic UI.

Environment Variables



Key:

Value:

REACT_APP_GITHUB...

Not Specified...

GITHUB_PERSONAL...

Not Specified...

```

const RepositoryItem = ({ ... }) => (
  <div>
    <div className="RepositoryItem-title">
      ...

    <div>
      ...

      <Mutation
        mutation={WATCH_REPOSITORY}
        variables={{
          id,
          viewerSubscription: isWatch(viewerSubscription)
            ? VIEWER_SUBSCRIPTIONS.UNSUBSCRIBED
            : VIEWER_SUBSCRIPTIONS.SUBSCRIBED,
        }}
        optimisticResponse={{
          updateSubscription: {
            __typename: 'Mutation',
            subscribable: {
              __typename: 'Repository',

```



```

        id,
        viewerSubscription: isWatch(viewerSubscription)
          ? VIEWER_SUBSCRIPTIONS.UNSUBSCRIBED
          : VIEWER_SUBSCRIPTIONS.SUBSCRIBED,
      },
    },
  },
  update={updateWatch}
)
...
</Mutation>

...
</div>
</div>

...
</div>
);

```

src/Repository/RepositoryItem/index.js

Run the application in the end and watch a repository, the “Watch” and “Unwatch” label of the button changes immediately after you click it. This is because the optimistic response arrives synchronously, while the real response is pending and resolves later. Since the `__typename` meta field comes with every Apollo request, it includes those as well.

An additional benefit of the optimistic response is that it makes the count of watchers update optimistically too. The function used in the `update` prop is called twice now, the first time with the optimistic response, and the second with a response from GitHub’s GraphQL API. It makes sense to capture identical information in the optimistic response expected as a mutation results in the function passed to the `update` prop of the `Mutation` component. For instance, if we don’t pass the `id` property in the `optimisticResponse` object, the function passed to the `update` prop throws an error, because it can’t retrieve the repository from the cache without an identifier.

At this point, it becomes debatable whether or not the `Mutation` component becomes too verbose. Using the Render Props pattern co-locates the data layer even more to the view-layer than the Higher-Order Components. Also, one could argue it doesn’t co-locate the data-layer but inserts it into the view-layer. When optimizations like the `update` and `optimisticResponse` props are put into the Render Prop Component, it can become too verbose for a scaling application. I advise using techniques you’ve learned as well as your own strategies to keep your source code concise.


I see four different ways to solve this issue:

- Keep the declarations inlined (see: `optimisticUpdate`)
- Extracting the inlined declarations as a variable (see: `update`).
- Perform a combination of 1 and 2 whereas only the most verbose parts are extracted
- Use Higher-Order Components instead of Render Props to co-locate data-layer, instead of inserting it in the view-layer

The first three are about **inserting** a data-layer into the view-layer, while the last is about **co-locating** it. Each comes with drawbacks. Following the second way, you might see yourself declaring functions instead of objects, or higher-order functions instead of functions because you need to pass arguments to them. With the fourth, you could encounter the same challenge in keeping HOCs concise. Then you could use the other three ways too but this time in a HOC rather than a Render Prop.

Confirm your [source code for the last section](#).

Check out the code below where we have already implemented the optimistic UIs for the star and unstar mutations:

Environment Variables 

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
import React from 'react';

import Link from '../Link';

import './style.css';

const Footer = () => (
  <div className="Footer">
    <div>
      <small>
        <span className="Footer-text">Built by</span>{' '}
        <Link
          className="Footer-link"
          href="https://www.robinwieruch.de"
        >
          Robin Wieruch
        </Link>{' '}
        <span className="Footer-text">with &hearts;</span>
      </small>
    </div>
  </div>
);
```

```

    </small>
  </div>
  <div>
    <small>
      <span className="Footer-text">
        Interested in GraphQL, Apollo and React?
      </span>{' '}
      <Link
        className="Footer-link"
        href="https://www.getrevue.co/profile/rwieruch"
      >
        Get updates
      </Link>{' '}
      <span className="Footer-text">
        about upcoming articles, books &
      </span>{' '}
      <Link className="Footer-link" href="https://roadtoreact.com">
        courses
      </Link>
      <span className="Footer-text">.</span>
    </small>
  </div>
</div>
);

export default Footer;

```

Exercises

1. If you are running this application locally, you can throttle your internet connection (often browsers offers such functionality) and experience how the `optimisticResponse` takes the `update` function into account even though the request is slow
2. Try different ways of co-locating or inserting your data-layer with render props and higher-order components.

Reading Tasks

1. Read more about [Apollo Optimistic UI in React with GraphQL](#).