Local State Management with Apollo Client in React

In this lesson, you will learn how to maintain and manage to update your local state using GraphQL Fragments and Apollo Client in React!

WE'LL COVER THE FOLLOWING

- Behind the scenes of “starring” a repository
- Updating the Count of Stars
- Using GraphQL Fragments
- Reading Tasks

Behind the scenes of “starring” a repository

Let’s get back to the `Repository` component. You have experienced that the `viewerHasStarred` boolean updates in the Apollo Client’s cache after a mutation was successful. That’s great, because Apollo Client handles this for you, based on the mutation result.

If you have followed the code from the mutation lesson, you should probably see something like a toggling **“Star”** and **“Unstar”** label for the button. All of this happens because you returned the `viewerHasStarred` boolean in your mutation result. Apollo Client is clever enough to update the repository entity, which is normalized accessible in the cache. That’s the powerful default behavior, isn’t it? You don’t need to handle the local state management yourself since Apollo Client figures it out for you as long as you provide useful information in the mutation’s result.

Updating the Count of Stars

However, Apollo Client doesn’t update the count of stars after the mutation. Normally, it is assumed that the count of stars increments by one when it is starred, with the opposite for unstarring. Since we don’t return a count of stargazers in the mutation result, you have to handle the update in Apollo

Client's cache yourself.

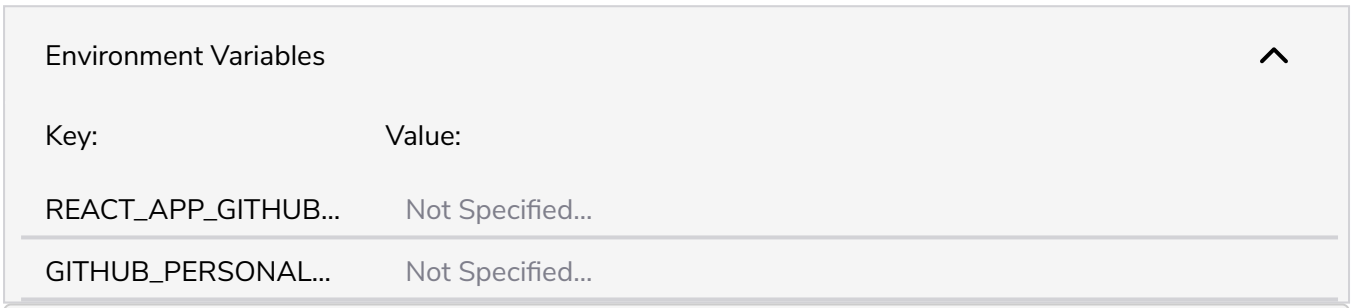
Using Apollo Client's `refetchQueries` option is the naive approach for a mutation call, or we can make use of a `Mutation` component to trigger a refetch for all queries where the query result might be affected by the mutation. But that's not the best way to deal with this problem. This is because it costs another query request to keep the data consistent after a mutation. In a growing application, this approach will eventually become problematic.

Fortunately, the Apollo Client offers other functionalities to read/write manually from/to the cache locally without more network requests. The `Mutation` component offers a prop where you can insert update functionality that has access to the Apollo Client instance for the update mechanism.

Using GraphQL Fragments

Before implementing the update functionality for the local state management, let's refactor another piece of code that will be useful for a local state update mechanism. The query definition next to your `Profile` component has grown to several fields with multiple object nestings. Previously, you learned about **GraphQL fragments**, and how they can be used to split parts of a query to reuse later.

Now, we will split all the field information you used for the repository's node. You can define this fragment in the *src/Repository/fragments.js* file to keep it reusable for other components.




```
import gql from 'graphql-tag';

const REPOSITORY_FRAGMENT = gql`
  fragment repository on Repository {
    id
    name
    url
    descriptionHTML
    primaryLanguage {
      name
    }
```

```
    owner {
      login
      url
    }
    stargazers {
      totalCount
    }
    viewerHasStarred
    watchers {
      totalCount
    }
    viewerSubscription
  }
`;

export default REPOSITORY_FRAGMENT;
```

src/Repository/fragment.js

We split this partial query (fragment) because it is used more often in this application in the next lessons for local state update mechanism, hence we go with the previous refactoring.

The fragment shouldn't be imported directly from the *src/Repository/fragments.js* path to our `Profile` component, because the *src/Repository/index.js* file is the preferred entry point to this module.

Environment Variables

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

```
import RepositoryList from './RepositoryList';
import REPOSITORY_FRAGMENT from './fragments';

export { REPOSITORY_FRAGMENT };

export default RepositoryList;
```

src/Repository/index.js

Finally, we import the fragment in the `Profile` component's file to use it again.

Environment Variables

Key:	Value:

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

```
...

import RepositoryList, { REPOSITORY_FRAGMENT } from '../Repository';
import Loading from '../Loading';
import ErrorMessage from '../Error';

const GET_REPOSITORIES_OF_CURRENT_USER = gql`
  {
    viewer {
      repositories(
        first: 5
        orderBy: { direction: DESC, field: STARGAZERS }
      ) {
        edges {
          node {
            ...repository
          }
        }
      }
    }
  }

  ${REPOSITORY_FRAGMENT}
`;

...
```

src/Profile/index.js

The refactoring is done! Your query is now more concise, and the fragment, in its natural repository module can be reused for other places and functionalities.

Next, we will use `Mutation` component's `update` prop to pass a function which will update the local cache eventually.

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

```
...

const updateAddStar = (client, mutationResult) => {
  ...
};

const RepositoryItem = ({ ... }) => (
  <div>
```

```
    <div className="RepositoryItem-title">
      ...

      <div>
        {viewerHasStarred ? (
          ...
        ) : (
          <Mutation
            mutation={STAR_REPOSITORY}
            variables={{ id }}
            update={updateAddStar}
          >
            ...
          </Mutation>
        )}
      </div>
    </div>

    ...
  </div>
);

export default RepositoryItem;
```

src/Repository/RepositoryItem/index.js

The function is extracted as its own JavaScript variable, otherwise, it ends up too verbose in the `RepositoryItem` component when keeping it inlined in the `Mutation` component. The function has access to the Apollo Client and the mutation result in its arguments, and we need both to update data so we can destructure the mutation result in the function signature. If you don't know how the mutation result looks like, check the `STAR_REPOSITORY` mutation definition again, where you defined all fields that should appear in the mutation result. For now, the important part is the `id` of the to-be-updated repository.


```
const updateAddStar = (
  client,
  { data: { addStar: { starrable: { id } } } },
) => {
  ...
};
```

src/Repository/RepositoryItem/index.js

You could have passed the `id` of the repository to the `updateAddStar()` function, which was a higher-order function in the `Mutation` component's render prop child function. You already have access to the repository's identifier in the `Repository` component.

Now comes the most exciting part of this lesson. We can use the Apollo Client to read data from the cache, but also to write data to it. The goal is to read the starred repository from the cache, which is why we need the `id` to increment the count of its stargazers by one and then write the updated repository back to the cache. We got the repository by its `id` from the cache by extracting the repository fragment. We can use it along with the repository identifier to retrieve the actual repository from Apollo Client's cache without querying all the data with a naive query implementation.

Environment Variables

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

```
...

import REPOSITORY_FRAGMENT from '../fragments';
import Link from '../../Link';
import Button from '../../Button';

...

const updateAddStar = (
  client,
  { data: { addStar: { starrable: { id } } } },
) => {
  const repository = client.readFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
  });

  // update count of stargazers of repository

  // write repository back to cache
};
```

src/Repository/RepositoryItem/index.js

The Apollo Client's cache that we set up to initialize the Apollo Client

normalizes and stores the queried data. Otherwise, the repository would be a deeply nested entity in a list of repositories for the query structure used in the `Profile` component. Normalization of a data structure makes it possible to retrieve entities by their identifier and their GraphQL `__typename` meta field. The combination of both is the default key, which is called a composite key, to read or write an entity from or to the cache. You may find out more about changing this default composite key in the exercises of this lesson.

Furthermore, the resulting entity has all properties specified in the fragment. If there is a field in the fragment not found on the entity in the cache, you may see the following error message: `Can't find field __typename on object …`. That's why we use the identical fragment to read from the local cache to query the GraphQL API.

After you have retrieved the repository entity with a fragment and its composite key, we will update the count of stargazers and write back the data to our cache. In this case, we will just increment the number of stargazers.


```
const updateAddStar = (
  client,
  { data: { addStar: { starrable: { id } } } },
) => {
  const repository = client.readFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
  });

  const totalCount = repository.stargazers.totalCount + 1;

  client.writeFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
    data: {
      ...repository,
      stargazers: {
        ...repository.stargazers,
        totalCount,
      },
    },
  });
};
```

src/Repository/RepositoryItem/index.js

Let's recap all three steps here:

- Firstly, we have retrieved (read) the repository entity from the Apollo Client using an identifier and the fragment;
- Secondly, we updated the information of the entity;

*Thirdly, we wrote back the data with updated information but kept all remaining information intact using the JavaScript spread operator. This is a manual update mechanism that can be used when a mutation is missing data.

It is a good practice to use an identical fragment for all three parts: the initial query, the `readFragment()`, and `writeFragment()` cache method. Your data structure for the entity stays consistent in your cache. For instance, if you forget to include a property defined by the fragment's fields in data object of the `writeFragment()` method, you get a warning: `Missing field __typename in …`.

On an implementation level, you learned about extracting fragments from a query or mutation. Fragments allow you to define your shared entities by GraphQL types. You can reuse those in your queries, mutations or local state management methods to update the cache. On a higher level, you learned that Apollo Client's cache normalizes your data, so you can retrieve entities that were fetched with a deeply nested query using their type and identifier as a composite key. Without it, you'd have to perform normalizations for all the fetched data before putting it in your store/state.

Confirm your source code for the last section.

Run the code below where we have implemented local cache updates for all the other mutations such as `removeStar` (decreasing the stargazers) and `updateSubscription` from the previous lessons:

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>
        <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;
```

Reading Tasks

1. Read more about Local State Management in Apollo Client
2. Read more about Fragments in Apollo Client
3. Read more about Caching in Apollo Client and the composite key to identify entities