Apollo local state and reactive variables

Apollo Client for State Management
Apollo client 3 enables the management of a local state through incorporating field policies and reactive variables.

Field policies define local fields so that the field is populated with information stored anywhere, like local storage or in-memory-storage and reactive variables.

Apollo Client (version >=3) provides two mechanisms to handle local state:

- Local-only fields and field policies
- Reactive variables

What is the Apollo client?
The Apollo client connects React App, GraphQL API, and besides is a state management library.

It helps to connect React App with GraphQL API. It provides methods to communicate with API, cache mechanisms, helpers, etc.

Apollo client provides an integrated state management system that allows to manage the state of whole application.

What is Apollo State?
Apollo Client has its state management system using GraphQL to communicate directly with external servers and provide scalability.

Apollo Client supports managing the local and remote state of applications, and able to interact with any state on any device with the same API.

Local-only fields and field policies
This mechanism allows to create client schema. Then, define field policies that describe wherefrom data came from. We can use Apollo cache or local storage.

Local state example
to handle local data inside a standard GraphQL query, use a @client directive for local fields:

```
query getMissions ($limit: Int!){
    missions(limit: $limit) {
        id
        name
        twitter
        website
        wikipedia
        links @client // this field is local
    }
}

```

Define local state using local-only fields

InMemory cache from Apollo
Apollo client provides a caching system for local data. Normalized data is saved in memory.

Field type polices
We can read and write to Apollo client cache and also can specify read, write, and merge functions and add custom logic there.

To define a local state,

1. Define field policy and pass it to the InMemoryCache
2. Add field to the query with @client directive

Install apollo client

```
npm install @apollo/client graphql
```

Initialize Apollo client
Import apollo client stuff in index.js:

```
import {
 ApolloClient,
 InMemoryCache,
 ApolloProvider,
} from "@apollo/client";

```

Create client instance

```
const client = new ApolloClient({
 uri: 'https://api.spacex.land/graphql/',
 cache: new InMemoryCache()
});;

```

API.spacex.land/graphql is a free public demo of GraphQL API.

To explore that API, copy the URL to the browser: https://api.spacex.land/graphql/

Connect Apollo with React by wrapping App component with ApolloProvider:

```
<ApolloProvider client={client}>
     <App />
</ApolloProvider>

```

ApolloProvider takes the client argument, which is our already declared Apollo Client.

We can use Apollo Client features in the App component and every child component.

The query for missions data
Let's get some data from the API. I want to get missions:

```
query getMissions ($limit: Int!){
  missions(limit: $limit) {
    id
    name
    twitter
    website
```

```
    wikipedia
  }
}

```

Results for this query when I passed 3 as a limit variable:

```
{
  "data": {
    "missions": [
      {
        "id": "9D1B7E0",
        "name": "Thaicom",
        "twitter": "https://twitter.com/thaicomplc",
        "website":
"http://www.thaicom.net/en/satellites/overview",
        "wikipedia": "https://en.wikipedia.org/wiki/Thaicom"
      },
      {
        "id": "F4F83DE",
        "name": "Telstar",
        "twitter": null,
        "website": "https://www.telesat.com/",
        "wikipedia": "https://en.wikipedia.org/wiki/Telesat"
      },
      {
        "id": "F3364BF",
        "name": "Iridium NEXT",
        "twitter":
"https://twitter.com/IridiumBoss?lang=en",
        "website": "https://www.iridiumnext.com/",
        "wikipedia":
"https://en.wikipedia.org/wiki/Iridium_satellite_constellation
"
      }
    ]
  }
}

```

Let's create a React component that receives that data and, for now, displays the name of the Mission on the screen.

Add Component: src/components/Missions/Missions.js

```
import React from "react"

export const Missions = () => {
   return <div>
       Missions component should be here.
   </div>
}

export default Missions;

```

Let's re-export component in src/components/Missions/index.js

```
export { default } from './Missions';
```

We need to query for data using the useQuery hook provided by the Apollo client.

In unit tests, need to have a component wrapped by **ApolloProvider**. For testing purposes, Apollo provides a unique Provider: **MockedProvider,** and it allows to add some mock data. Let's use it.

```
// src/components/Missions/__tests__/Missions.spec.js

Import MockedProvider:

import { MockedProvider } from '@apollo/client/testing';

```

Create the file queries/missions.gql.js with the following content:

```
import { gql } from '@apollo/client';

export const GET_MISSIONS = gql`
   query getMissions ($limit: Int!){
       missions(limit: $limit) {
           id
           name
           twitter
           website
            wikipedia
```

```
        }
    }
`;

```

First import useQuery hook, and GET_MISSIONS query in src/components/Missions/Missions.js

```
import { useQuery } from "@apollo/client";
import { GET_MISSIONS } from "../../queries/missions.gql";

```

Let's query for the data in the component body:

```
const  { data } = useQuery(GET_MISSIONS, {
       variables: {
           limit: 3
       }
});

```

```
return shouldDisplayMissions;
The full component code:
import React, { useMemo } from "react"
import { useQuery } from "@apollo/client";
import { GET_MISSIONS } from "../../queries/missions.gql";

export const Missions = () => {
   const  { data } = useQuery(GET_MISSIONS, {
       variables: {
           limit: 3
       }
   });

   const shouldDisplayMissions = useMemo(() => {
       if (data?.missions?.length) {
           return data.missions.map(mission => {
               return <div key={mission.id}>
                   <h2>{mission.name}</h2>
               </div>
           })
       }

       return <h2>There are no missions</h2>
   }, [data]);

   return shouldDisplayMissions;
```

```
27}
28
29export default Missions;
30
```

The last thing for this step is to inject components into the app and see missions in the browser.

```
1// App.js
2
3import Missions from './components/Missions';
4
5function App() {
6 return <Missions/>;
7}
8
9export default App;
10
```

First, grab the loading flag from the useQuery hook results:

```
1const  { data, loading } = useQuery(GET_MISSIONS, {
2    variables: {
3        limit: 3
4    }
5});
6
```

Add loading indicator:

```
1if (loading) {
2    return <p>Loading...</p>
3}
4
5return shouldDisplayMissions;
6

1//src/components/Missions/Missions.module.css
2.mission {
3   border-bottom: 1px solid black;
4   padding: 15px;
5}
6
```

Now, import CSS module in Missions.js file:

```
1import classes from './Missions.module.css'
2
```

and add mission class to the mission div:

```
1return <div key={mission.id} className={classes.mission}>
```

```
2     <h2>{mission.name}</h2>
3</div>
4
5
```

Here is the result: