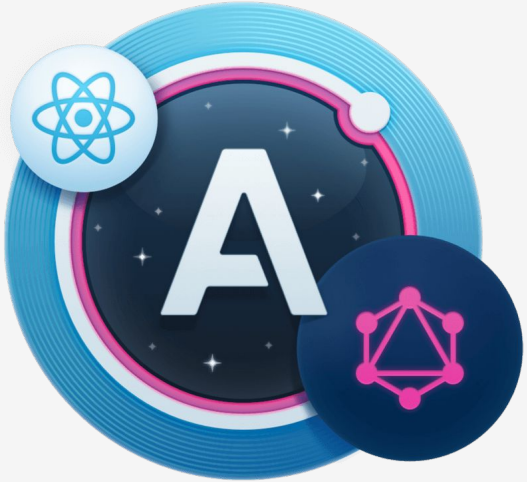
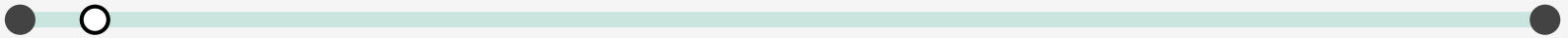


Error Handling With React & GraphQL

What this talk is about

1. Different types of GraphQL errors
2. Best practices to deal with those errors
3. Practical use-cases and examples using Apollo Client 2.0



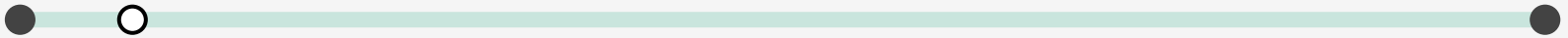
Contents

Refer this to know where you are.

Why Error handling is hard in GraphQL?

1. GraphQL spec itself provides little guidance on how to format error responses, requiring only a **message** field with a string description of the error.

2. We don't have **HTTP** like status code in GraphQL Errors



```
{
  "errors": [
    {
        "message": "GraphQL error: You must be logged in"
        "locations": [],
        "path": [
            "protectedAction"
        ],
    }
  ]
}
```

This is how GraphQL Error message looks Like

What types of GraphQL errors we can expect?

1. Server problems (**5xx HTTP codes**, 1xxx WebSocket codes)
2. Client problems e.g. rate-limited, unauthorized, etc. (**4xx HTTP codes**)
3. The query is missing/malformed
4. The user-supplied variables or context is bad and the resolve/subscribe function intentionally throws an error
5. And many more...

We need a way to modify GraphQL messages to include **HTTP** like status **codes**

```
{
  "errors": [
    {
        "code": "UNAUTHENTICATED",
        "message": "GraphQL error: You must be logged in",
        "locations": [],
        "path": [
            "protectedAction"
        ]
    }
  ]
}
```

With every error message a **code** i.e. (String constant) is included. It can be used to identify type of error

```
{
  "errors": [
    {
        "status": 200,
        "message": "GraphQL error: You must be logged in",
        "locations": [],
        "path": [
            "protectedAction"
        ]
    }
  ]
}
```

Or with every error message standard **HTTP** status code can be used

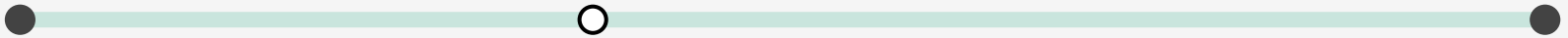
Wait..., what about GraphQL Specs..?

GraphQL does not encourages to add extra fields to error messages

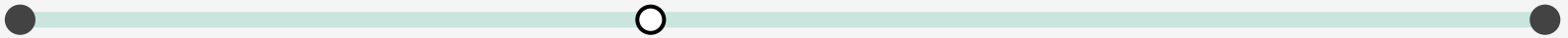
So out of two which one pattern should i choose ?
Totally depends upon you or bring your own...

DEMO

(Lets see a GraphQL Server in action)



Apollo Client, A GraphQL client library



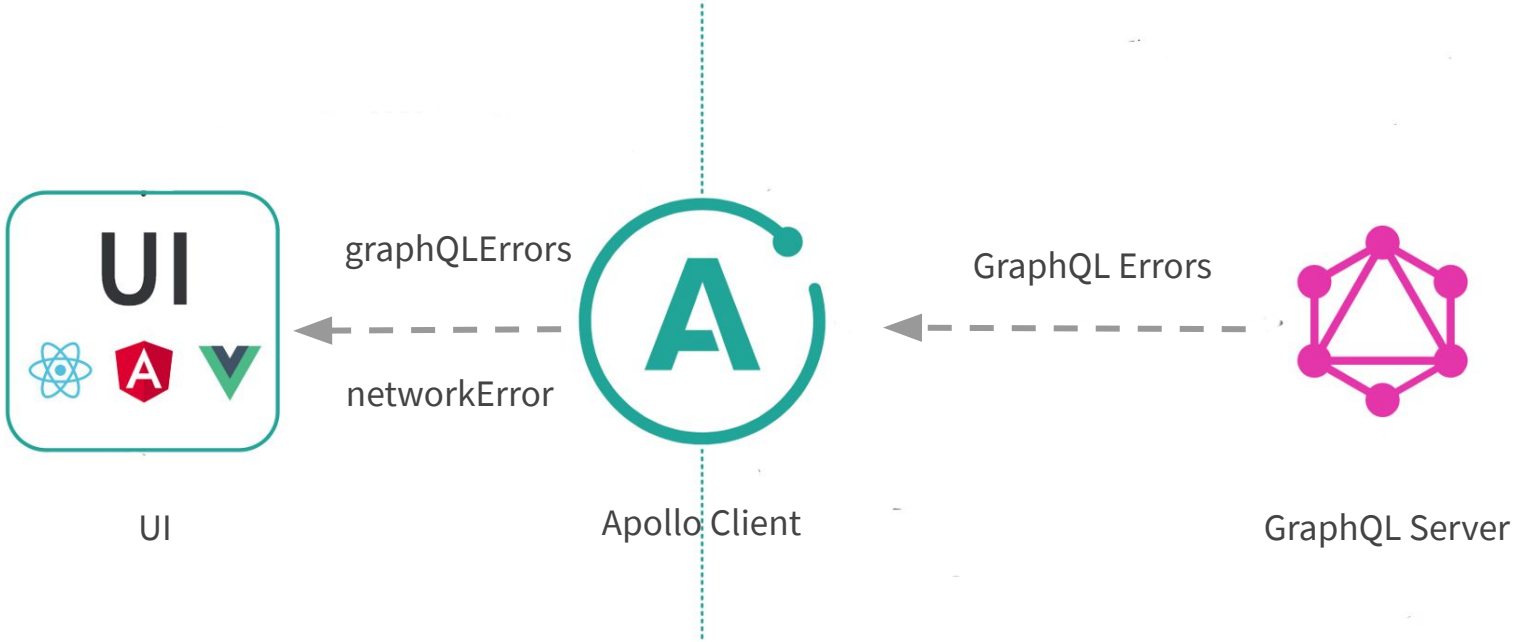
How Apollo Client can help us

1. **Declarative data fetching**
   With Apollo's declarative approach to data fetching, all of the logic for retrieving your data, tracking loading and error states

2. **Zero-config caching**
   Just by setting up Apollo Client, you get an intelligent cache out of the box with no additional configuration required.

3. **Universally compatible**
   Apollo works with any build setup, any GraphQL server, and any GraphQL schema

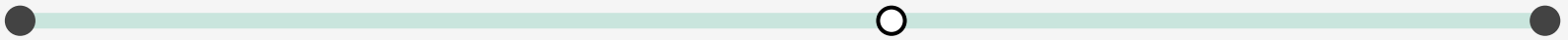
Error handling by Apollo Client



graphQLErrors vs networkError

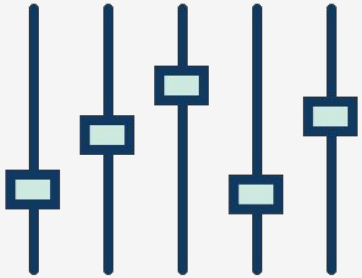
1. **networkError**: Errors that are thrown outside of your resolvers. For example, the client failed to connect to your GraphQL endpoint

2. **graphQLErrors**: Any error that is thrown within your resolvers code



```
{
  "error": {
    "graphQLErrors": [
      {
        "message": "forbidden",
        "locations": [],
        "path": [
          "protectedAction"
        ],
        "code": 403,
      }
    ],
    "networkError": null,
    "message": "GraphQL error: You must be logged in"
  }
}
```

How GraphQL Error looks at Client Side



Now let's do actual error handling

So far we have modified GraphQL Error messages to include error status codes.
We also understood how Apollo Client handles the errors

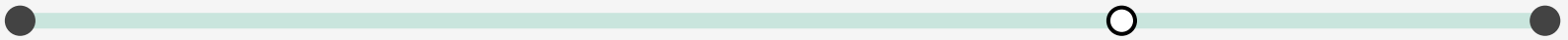
Application vs Query Level error handling

With Apollo Client you can split error handling logic into two parts. Application level errors an Query level error

Application vs Query Level error handling

1. **Application Level**: Application level errors can be handled at this stage. Such as permission errors, network errors, authentication errors etc.

2. **Query Level**: Query level errors can be handled at this stage. Such as malformed query. Query could not be resolved, resolver thrown a graphql error



```
import ApolloClient from 'apollo-boost';

const client = new ApolloClient({
  uri: '<your graphql endpoint>',
  onError: ({ graphQLErrors, networkError, operation, forward }) => {
    if (graphQLErrors) {
      for (let err of graphQLErrors) {
        // handle errors differently based on its error code
        switch (err.extensions.code) {
          case 403:
           // Your error handling logic
           //  ...

          // handle other errors
          case 500':
            //  ...
        }
      }
    }
  },
});
```

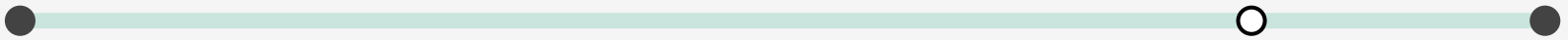
Application Level error handling

```
<Query query={GET_DOGS}>
  {({ loading, error, data }) => {

    if (loading) return "Loading...";

    if (error) return `Error! ${error.message}`;

    return (
      <pre>
       {JSON.stringify(data)}
      </pre>
    );
  }}
</Query>
```

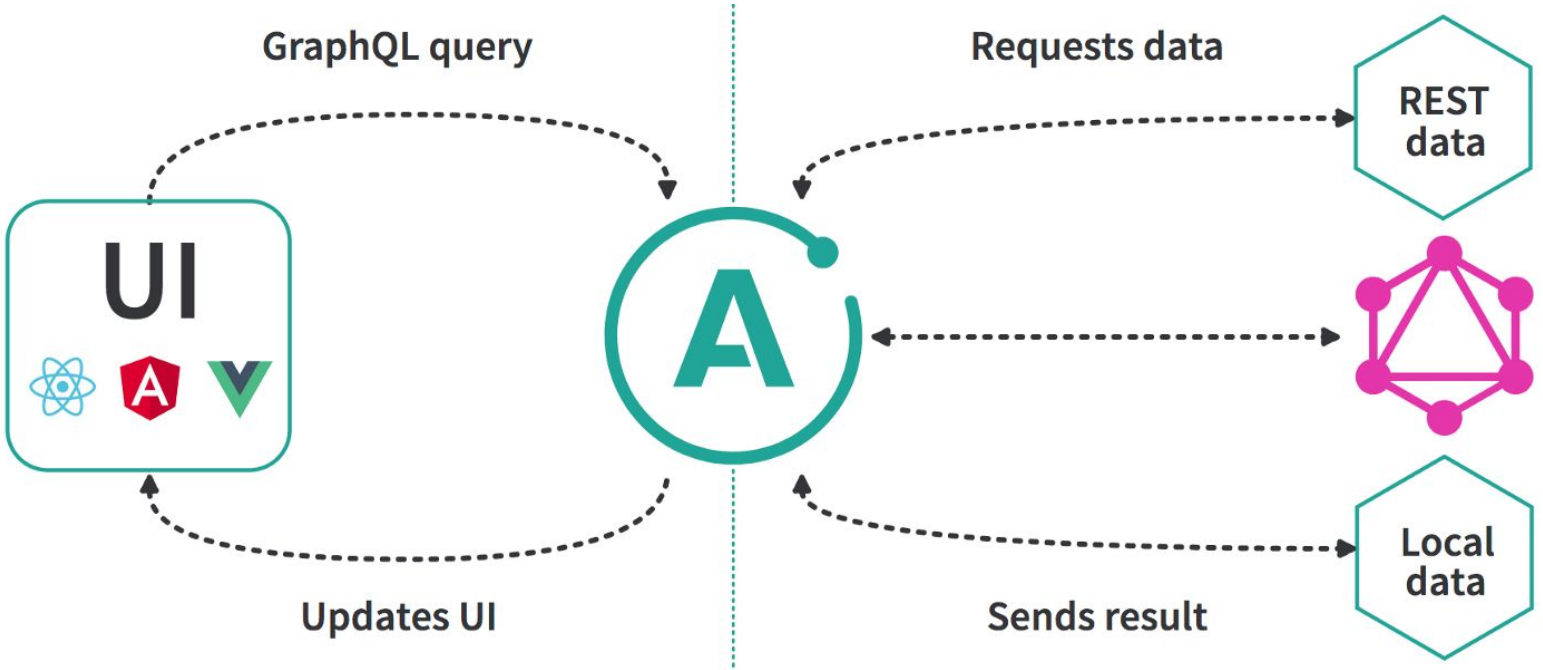
Query Level error handling

How it works

1. All errors are first passed through the **Application Level** error handler
2. **Application Level** error handler can take action depending upon the status code
3. If none of the case mathes error is forwarded to the **Query Level** error handler
4. **Query Level** error handler can take action depending upon the status code.



Apollo Client under the hood

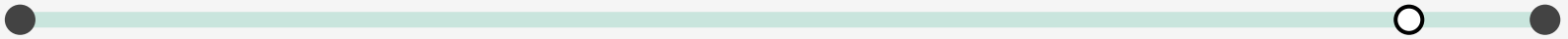


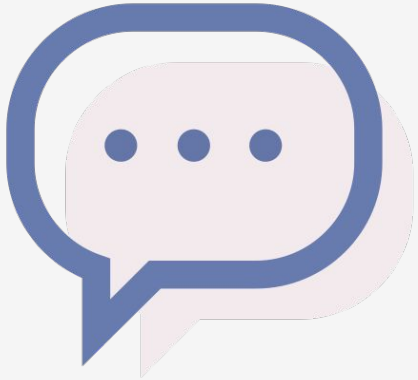
DEMO

(Let's see Apollo Client in action)

Recap

1. Alongside with the **GraphQL Error Messages**, also include the error status
2. While modifying the message be careful about **GraphQL Specs**
3. Write your error handling logic at client side based on the **error status** in the **error messages**





Questions...?

Contact me

Gufran Mirza

Software engineer @continuum

@_imGufran

@gufranmirza

@gufranmirza

ednsquare.com/@gufranmirza