Apollo Client Presentation

< Presented By Benmoussa Younes >

OmranSoftware - 2023

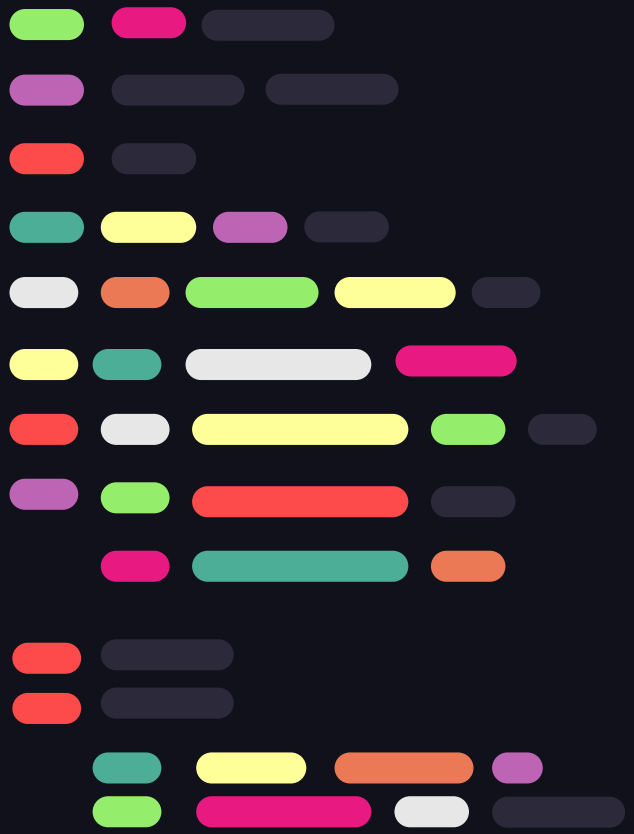
Table of contents

Table of contents

Table of contents

Table of contents

Table of contents



Part 01

01 { ..

Introduction to Apollo Client

< Apollo Client is a comprehensive state management library for JavaScript >

} ..

Apollo Client Features

Declarative data fetching

Write a query and receive data without manually tracking loading states.

Excellent developer experience

Enjoy helpful tooling for TypeScript, Chrome / Firefox devtools, and VS Code.

Designed for modern React

Take advantage of the latest React features, such as hooks.

Apollo Client Features

Incrementally adoptable

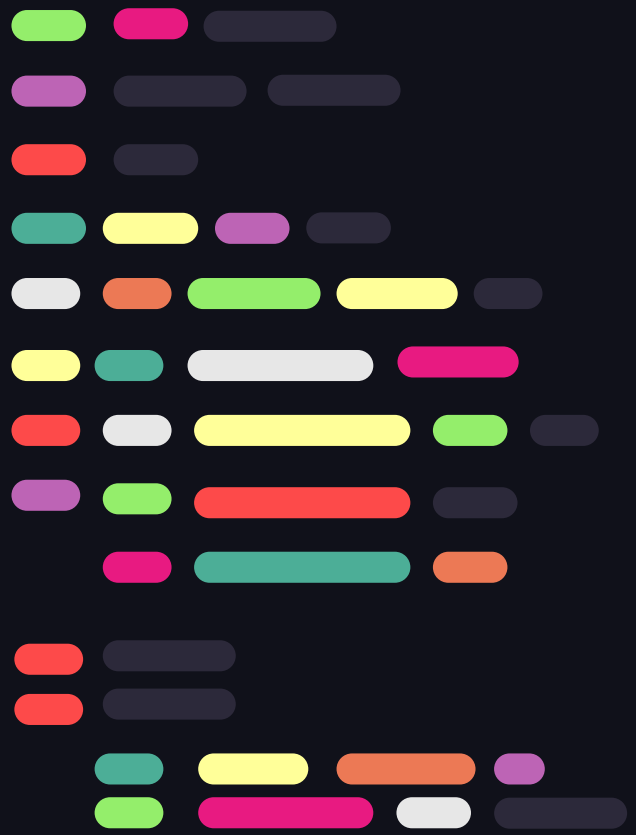
Drop Apollo into any JavaScript app and incorporate it feature by feature.

Universally compatible

Use any build setup and any GraphQL API.

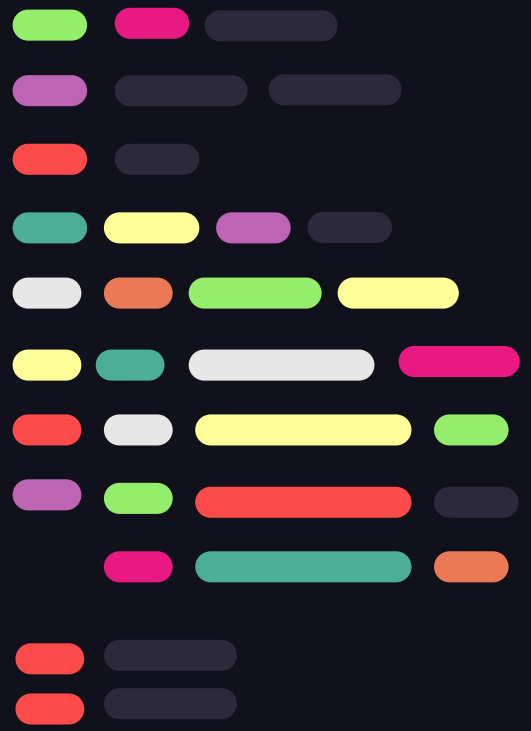
Community driven

Share knowledge with thousands of developers in the GraphQL community.



Why Apollo Client ?

Why Apollo Client ?



Apollo Client is a state management library that simplifies managing remote and local data with GraphQL

- Declarative approach to data fetching
- Intelligent caching
- Custom functionality

Declarative data fetching

```
function ShowBooks() {
    const { loading, error, data } = useQuery(GET_BOOKS);
    //
    if (error) return <Error />;
    if (loading) return <Fetching />;
    //
    return <DogList dogs={data.books} />;
  }
```

Combining local & remote data

< Apollo Client includes local state management features enabling you to use your Apollo cache as the single source of truth for your application's data. >

< By using Apollo Client's local state functionality, you can include local fields and remotely fetched fields in the same query >

Combining local & remote data

```
const GET_BOOK = gql`
  query GetBookByPaper($paper: String!) {
    paper(paper: $paper) {
      images {
        url
        id
        isLiked @client
      }
    }
  }
`;
```

Zero-config caching

< One of the key features that sets Apollo Client apart from other data management solutions is its local, in-memory, normalized cache. >

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

const client = new ApolloClient({
  cache: new InMemoryCache(),
});
```

Zero-config caching

< One of the key features that sets Apollo Client apart from other data management solutions is its local, in-memory, normalized cache. >

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

const client = new ApolloClient({
  cache: new InMemoryCache(),
});
```

Practical Example 01

The below query, GET_ALL_DOGS, fetches a list of dogs and information about each dog:

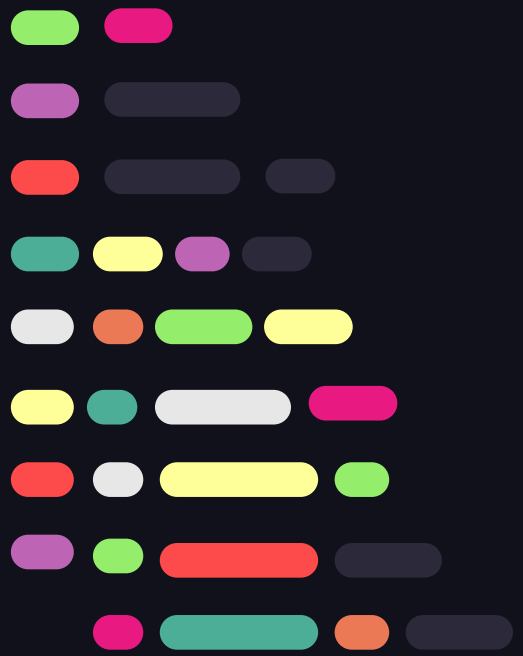
```
const GET_ALL_DOGS = gql`
  query GetAllDogs {
    dogs {
      id
      breed
      displayImage
    }
  }
`;
```

Practical Example 01

The below mutation, UPDATE_DISPLAY_IMAGE, updates a specified dog's displayImage and returns the updated dog:

```
const UPDATE_DISPLAY_IMAGE = gql`
  mutation UpdateDisplayImage($id: String!, $displayImage: String!)
{
    updateDisplayImage(id: $id, displayImage: $displayImage) {
      id
      displayImage
    }
  }
`;
```

Practical Example 01



When we run the UPDATE_DISPLAY_IMAGE mutation We want to ensure that our dog's image is updated everywhere in our application (same for any previously cached).

This Can be done Automatically done by just running the mutation query after the fetch query

Benefit : Avoid Refetching Information Already contained in our cache

Practical Example 01

When we run the UPDATE_DISPLAY_IMAGE mutation, we want to ensure that our dog's image is updated everywhere in our application. We also need to ensure we update any previously cached data about that dog.

Our UPDATE_DISPLAY_IMAGE mutation returns the object the mutation modified (i.e., the id and displayImage of the dog), enabling Apollo Client to automatically overwrite the existing fields of any previously cached object with the same id.

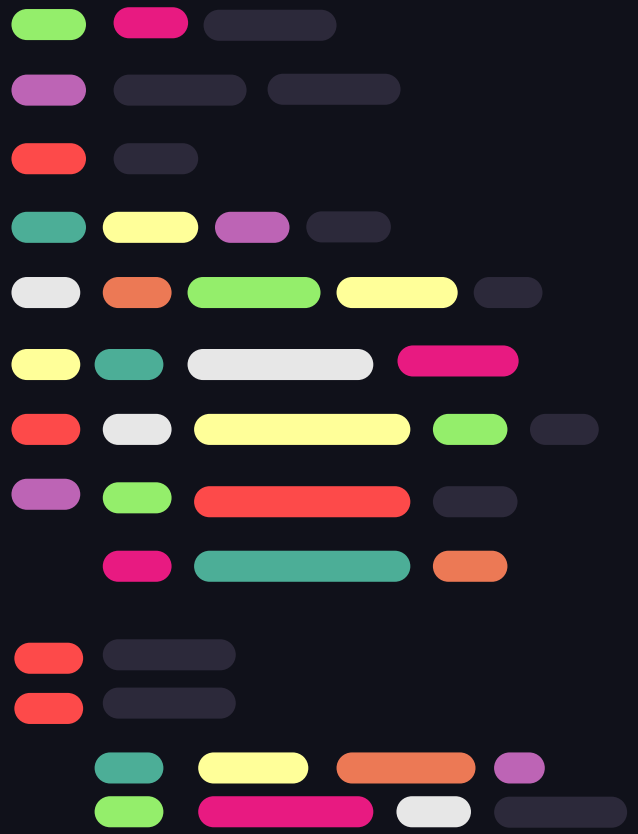
Tying it all together, if we've already run the GET_ALL_DOGS query before Apollo Client runs the UPDATE_DISPLAY_IMAGE mutation, it automatically updates the changed dog's displayImage in our local cache.

02 { ..

Fetching with Apollo Client

< Apollo Client enable fetching GraphQl data >

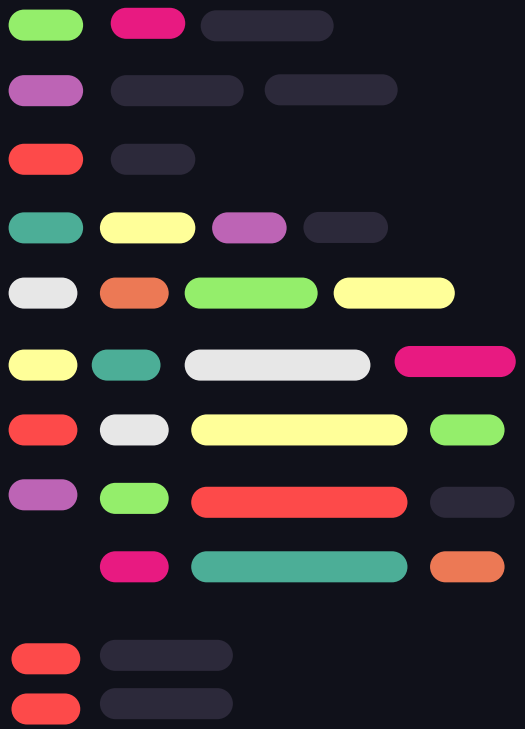
} ..



Manage Data

< manage both local and remote data with GraphQL.
Use it to fetch, cache, and modify application data, all while automatically updating your UI >

2.1 Manage Data



The useQuery React hook is the primary API for executing queries in an Apollo application.

useQuery returns an object from Apollo Client that contains loading, error, data properties you can use to render your UI.

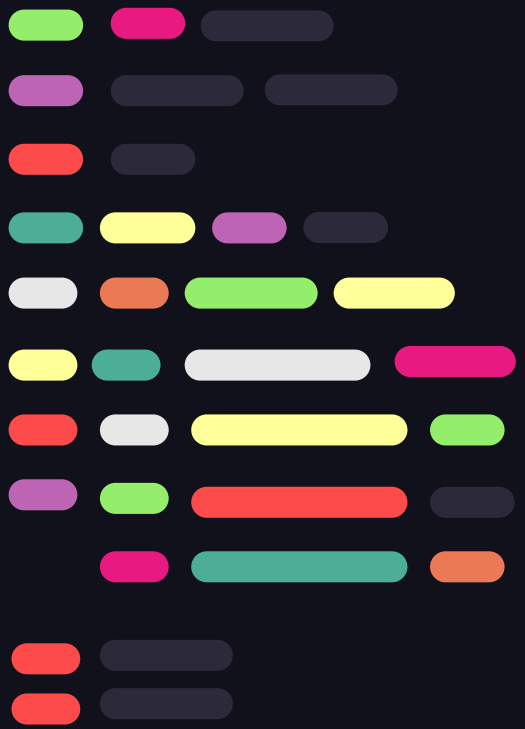
Practical Example 2.1

```
function Dogs({ onDogSelected }) {
  const { loading, error, data } = useQuery(GET_DOGS);

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

  return (
    <select name='dog' onChange={onDogSelected}>
      {data.dogs.map((dog) => (
        <option key={dog.id} value={dog.breed}>
          {dog.breed}
        </option>
      ))}
    </select>
  );
}
```

2.2 Caching query results



Whenever Apollo Client fetches query results from your server, it automatically caches those results locally. This makes later executions of that same query extremely fast.

To see this caching in action, let's build a new component called DogPhoto.

DogPhoto accepts a prop called breed that reflects the current value of the dropdown menu in our Dogs component

Practical Example 2.2

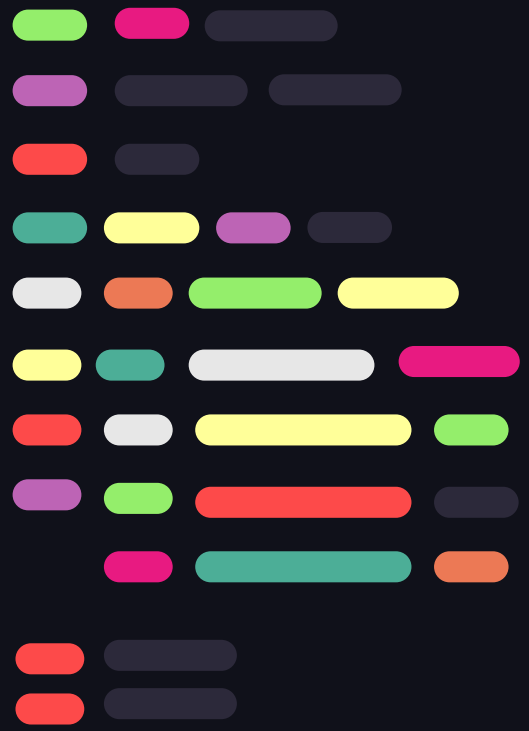
```
const GET_DOG_PHOTO = gql`
  query Dog($breed: String!) {
    dog(breed: $breed) {
      id
      displayImage
    }
  }
`;

function DogPhoto({ breed }) {
  const { loading, error, data } = useQuery(GET_DOG_PHOTO, {
    variables: { breed },
  });

  if (loading) return null;
  if (error) return `Error! ${error}`;

  return (
    <img src={data.dog.displayImage} style={{ height: 100, width: 100 }} />
  );
}
```

2.2 Caching query results

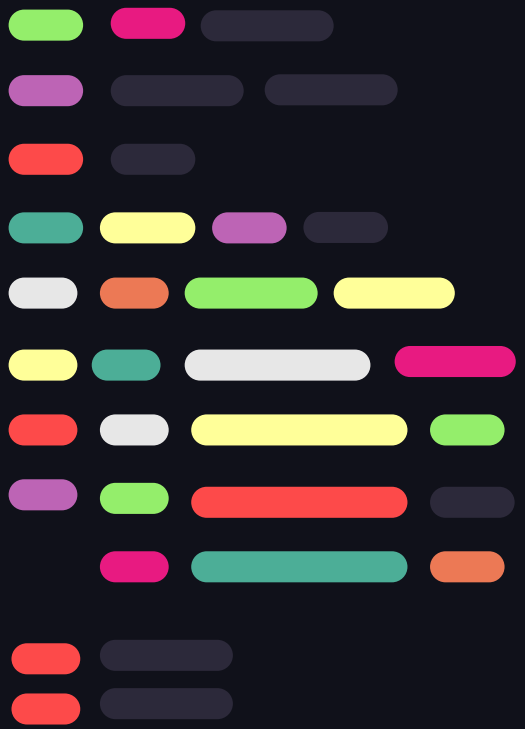


Select bulldog from the dropdown to see its photo appear.

Then switch to another breed, and then switch back to bulldog. You'll notice that the bulldog photo loads instantly the second time around. This is the cache at work!

Next, let's learn some techniques for ensuring that our cached data is fresh.

2.2 Caching query results



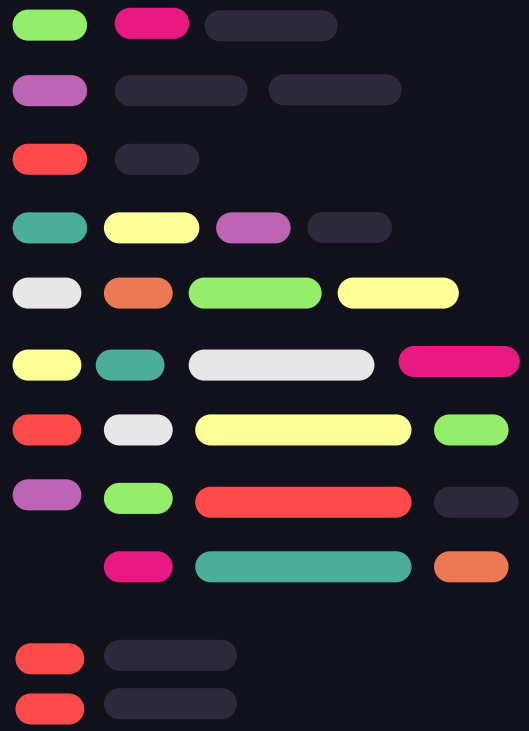
Remember Cirrus Cad Issue ?

UI was not updating on session change because of the default cache usage

We gonna see how to resolve that problem with different solutions on the coming titles

01 – A solution to that problem was simply updating the cache (we gonna see it in details on the cache presentation part)

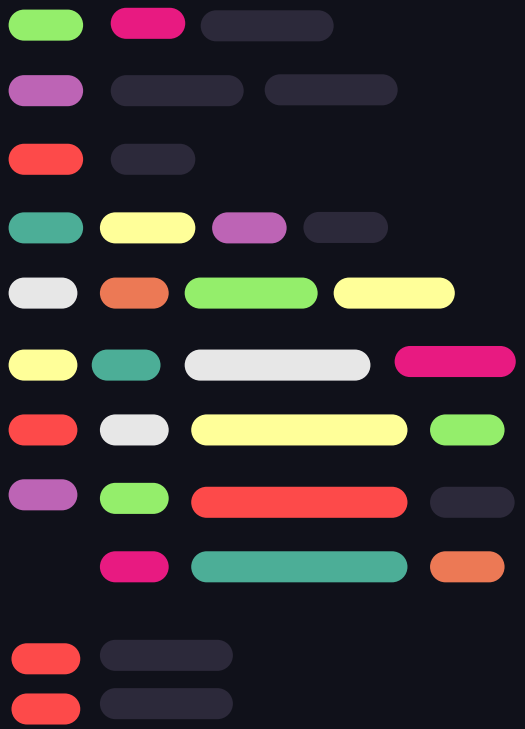
2.3 Updating cached query results



Sometimes, you want to make sure that your query's cached data is up to date with your server's data.

Apollo Client supports two strategies for this: polling and refetching.

2.3.1 Polling



Polling provides near-real-time synchronization with your server by executing your query periodically at a specified interval.

To enable polling for a query, pass a pollInterval configuration option to the useQuery hook with an interval in milliseconds

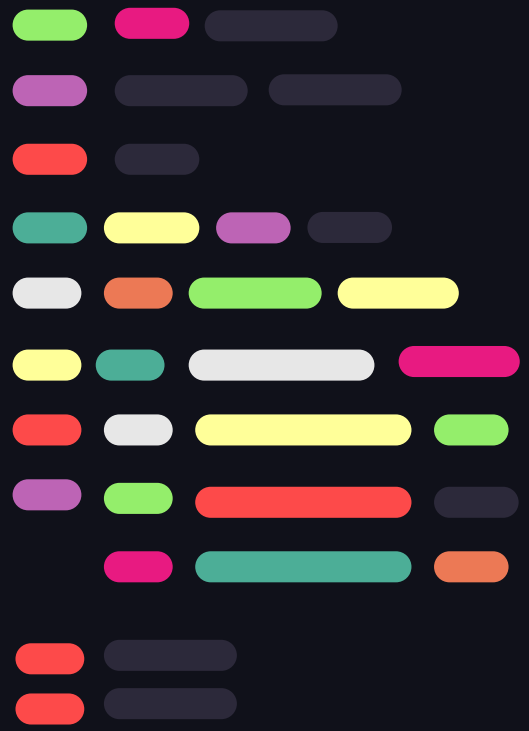
Practical Example 2.3.1

```
function DogPhoto({ breed }) {
  const { loading, error, data } = useQuery(GET_DOG_PHOTO, {
    variables: { breed },
    pollInterval: 500,
  });

  if (loading) return null;
  if (error) return `Error! ${error}`;

  return (
    <img src={data.dog.displayImage} style={{ height: 100, width: 100 }} />
  );
}
}
```

2.3.2 Refetching



Refetching enables you to refresh query results in response to a particular user action, as opposed to using a fixed interval (polling) .

Let's add a button to our DogPhoto component that calls our query's refetch function whenever it's clicked.

Practical Example 2.3.2

```
function DogPhoto({ breed }) {
  const { loading, error, data, refetch } = useQuery(GET_DOG_PHOTO, {
    variables: { breed },
  });

  if (loading) return null;
  if (error) return `Error! ${error}`;

  return (
    <div>
      <img src={data.dog.displayImage} style={{ height: 100, width: 100 }} />
      <button onClick={() => refetch({ breed: 'new_dog_breed' })}>
        Refetch new breed!
      </button>
    </div>
  );
}
```

2.3.2 Refetching

You can optionally provide a new variables object to the refetch function.

If you avoid passing a variables object and use only refetch(), the query uses the same variables that it used in its previous execution.

If you provide new values for some of your original query's variables but not all of them,
refetch uses each omitted variable's original value.

2.3.2 Refetching

Cirrus Cad Issue ?

Why using refetch did not resolve the issue ?

Refetch recognize that we provided a variable that has been already used for the Query `GET_ALL_SESSIONS_QUERY,`

So it automatically load cached data for that variable instead of refetching the data remotely

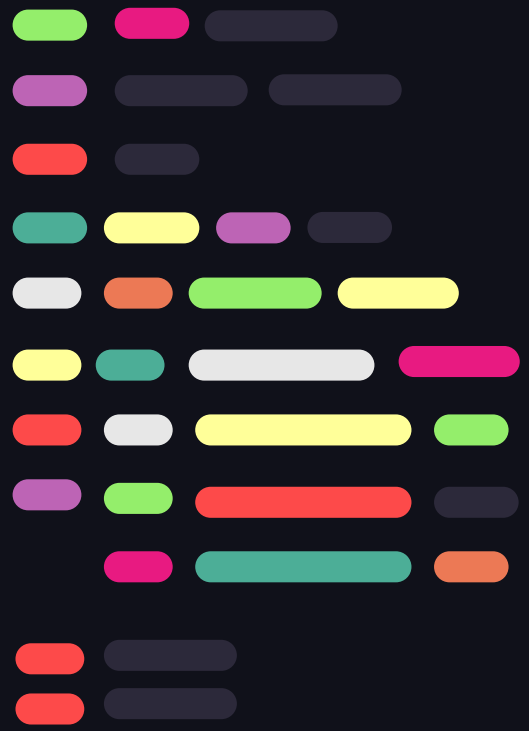
2.3.2 Refetching

Cirrus Cad Issue ?

03 - A solution to that problem

Refetching the GetSessionQuery

2.4 Inspecting loading states

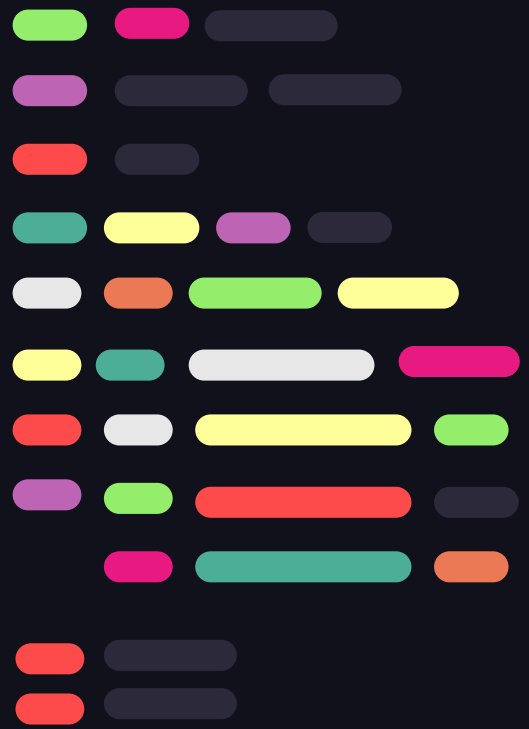


We've already seen that the useQuery hook exposes our query's current loading state.

This is helpful when a query first loads, but what happens to our loading state when we're refetching or polling ?

Let's return to our refetching example from the previous section .

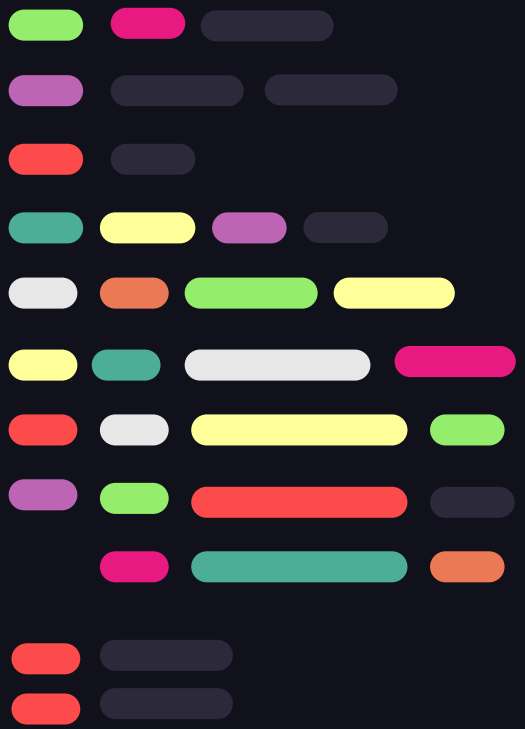
2.4 Inspecting loading states



- If you click the refetch button, you'll see that the component doesn't re-render until the new data arrives.

What if we want to indicate to the user that we're refetching the photo ?

2.4 Inspecting loading states



The useQuery hook's result object provides fine-grained information about the status of the query via the networkStatus property.

To take advantage of this information, we set the notifyOnNetworkStatusChange option to true

Practical Example 2.4

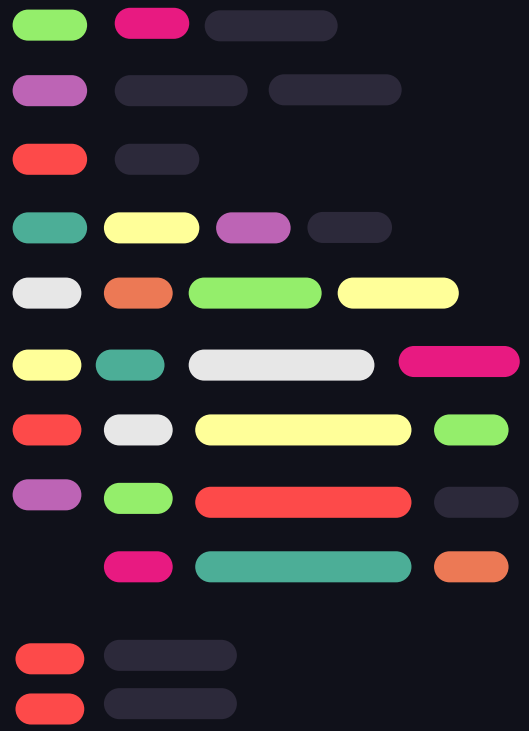
```
import { NetworkStatus } from '@apollo/client';

function DogPhoto({ breed }) {
  const { loading, error, data, refetch, networkStatus } = useQuery(
    GET_DOG_PHOTO,
    {
      variables: { breed },
      notifyOnNetworkStatusChange: true,
    }
  );

  if (networkStatus === NetworkStatus.refetch) return 'Refetching!';
  if (loading) return null;
  if (error) return `Error! ${error}`;

  return (
    <div>
      <img src={data.dog.displayImage} style={{ height: 100, width: 100 }} />
      <button onClick={() => refetch({ breed: 'new_dog_breed' })}>
        Refetch!
      </button>
    </div>
  );
}
```

2.5 Inspecting error states



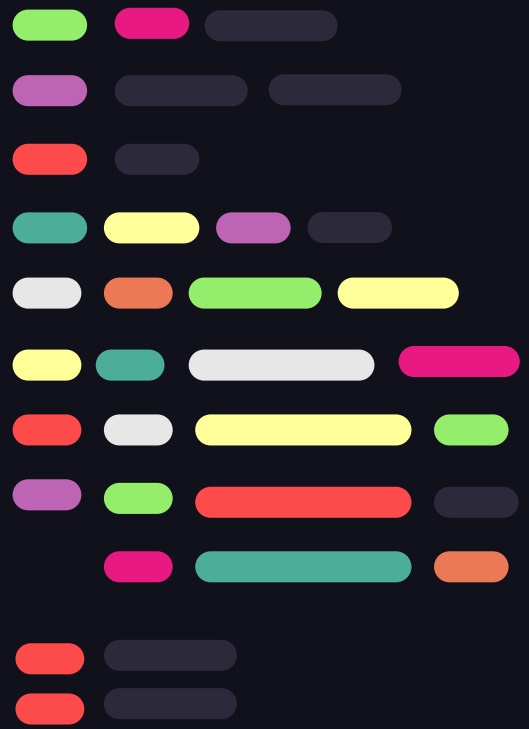
You can customize your query error handling by providing the errorPolicy configuration option to the useQuery hook.

The default value is none, which tells Apollo Client to treat all GraphQL errors as runtime errors.

2.5 Inspecting error states

POLICY	DESCRIPTION
none	If the response includes GraphQL errors, they are returned on error.graphQLErrors and the response data is set to undefined even if the server returns data in its response. This is the default error policy.
ignore	graphQLErrors are ignored (error.graphQLErrors is not populated), and any returned data is cached and rendered as if no errors occurred.
all	Both data and error.graphQLErrors are populated, enabling you to render both partial results and error information.

2.6 Manual execution with useLazyQuery



When React renders a component that calls useQuery, Apollo Client automatically executes the corresponding query.

But what if you want to execute a query in response to a different event, such as a user clicking a button ?

Practical Example 2.6

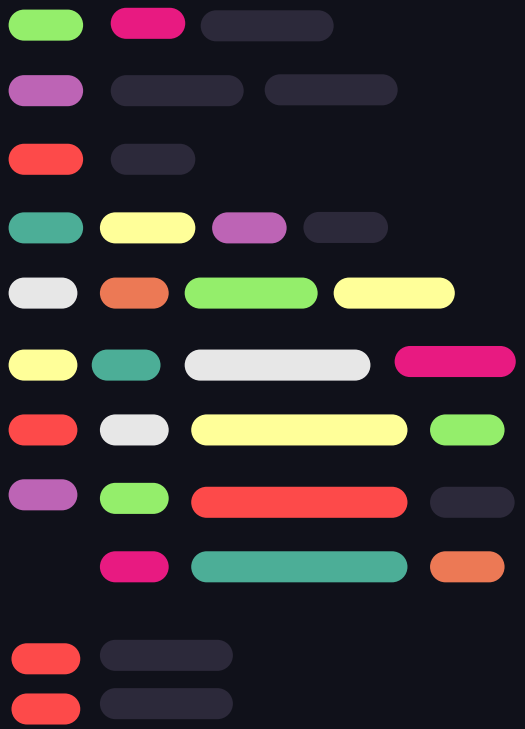
```
import React from 'react';
import { useLazyQuery } from '@apollo/client';

function DelayedQuery() {
  const [getDog, { loading, error, data }] = useLazyQuery(GET_DOG_PHOTO);

  if (loading) return <p>Loading ...</p>;
  if (error) return `Error! ${error}`;

  return (
    <div>
      {data?.dog && <img src={data.dog.displayImage} />}
      <button onClick={() => getDog({ variables: { breed: 'bulldog' } })}>
        Click me!
      </button>
    </div>
  );
}
```

2.7 Setting a fetch policy



By default, the useQuery hook checks the Apollo Client cache to see if all the data you requested is already available locally.

useQuery returns that data and doesn't query your GraphQL server.

This cache-first policy is Apollo Client's default fetch policy.

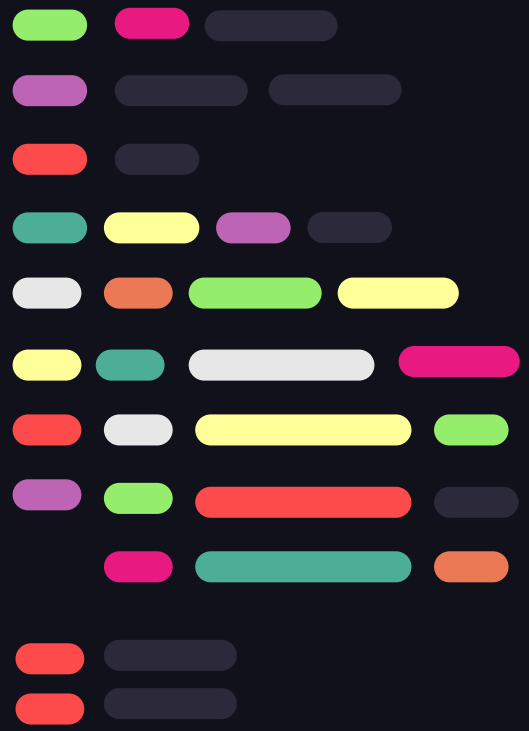
Practical Example 2.7

You can specify a different fetch policy for a given query.

To do so, include the fetchPolicy option in your call to useQuery:

```
const { loading, error, data } = useQuery(GET_DOGS, {
  fetchPolicy: 'network-only', // Doesn't check cache before making a network request
});
```

2.7.1 nextFetchPolicy



You can also specify a query's nextFetchPolicy.

If you do, fetchPolicy is used for the query's first execution, and nextFetchPolicy is used to determine how the query responds to future cache updates :

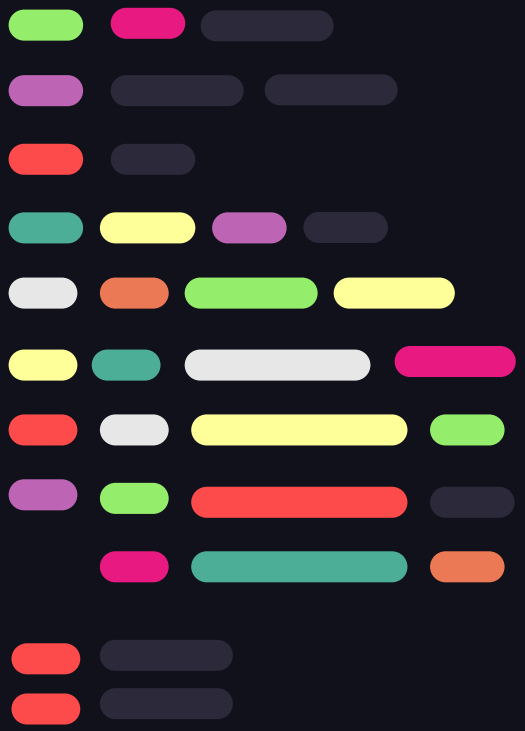
```
const { loading, error, data } = useQuery(GET_DOGS, {
  fetchPolicy: 'network-only', // Used for first execution
  nextFetchPolicy: 'cache-first', // Used for subsequent executions
});
}
```

2.7.2 nextFetchPolicy Fn

If you want to apply a single nextFetchPolicy by default, because you find yourself manually providing nextFetchPolicy for most of your queries

```
new ApolloClient({
  link,
  client,
  defaultOptions: {
    watchQuery: {
      nextFetchPolicy: 'cache-only',
    },
  },
});
```

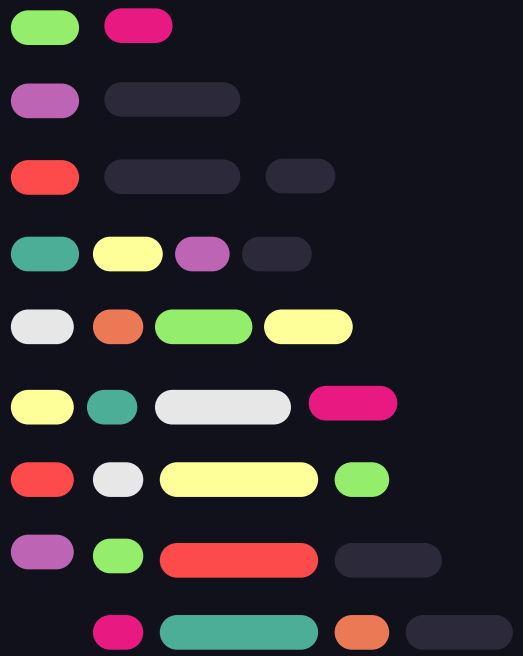
2.7.2 nextFetchPolicy Fn



If you want more control over how nextFetchPolicy behaves, you can provide a function instead of a WatchQueryFetchPolicy string

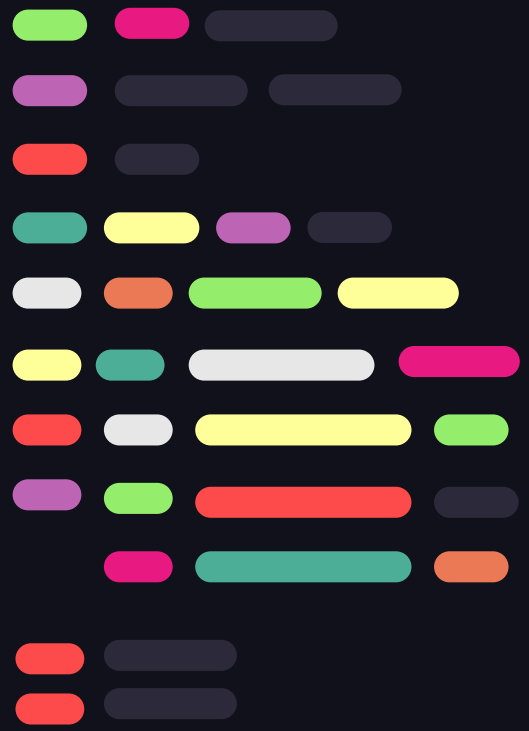
In addition to being called after each request, your nextFetchPolicy function will also be called when variables change, which by default resets the fetchPolicy to its initial value

Practical Example 2.7.2



```
new ApolloClient({
  link,
  client,
  defaultOptions: {
    watchQuery: {
      nextFetchPolicy(currentFetchPolicy) {
        if (
          currentFetchPolicy === 'network-only' ||
          currentFetchPolicy === 'cache-and-network'
        ) {
          // Demote the network policies (except "no-cache") to "cache-first"
          // after the first request.
          return 'cache-first';
        }
        // Leave all other fetch policies unchanged.
        return currentFetchPolicy;
      },
    },
  },
});
```

2.7.3 nextFetchPolicy Fn



To intercept and handle the variables-changed case yourself, you can use the NextFetchPolicyContext object passed as the second argument to your nextFetchPolicy function

In order to debug these nextFetchPolicy transitions, it can be useful to add console.log or debugger statements to the function body, to see when and why the function is called.

2.8 Supported fetch policies

NAME	DESCRIPTION
cache-first	Apollo Client first executes the query against the cache. If all requested data is present in the cache, that data is returned.
cache-only	Apollo Client executes the query only against the cache. It never queries your server in this case. A cache-only query throws an error if the cache does not contain data for all requested fields.
cache-and-network	Apollo Client executes the full query against both the cache and your GraphQL server. The query automatically updates if the result of the server-side query modifies cached fields.

2.8 Supported fetch policies

NAME	DESCRIPTION
network-only	Apollo Client executes the full query against your GraphQL server, without first checking the cache. The query's result is stored in the cache.
no-cache	Similar to network-only, except the query's result is not stored in the cache.
standby	Uses the same logic as cache-first, except this query does not automatically update when underlying field values change. You can still manually update this query with refetch and updateQueries.

2.9 useQuery API

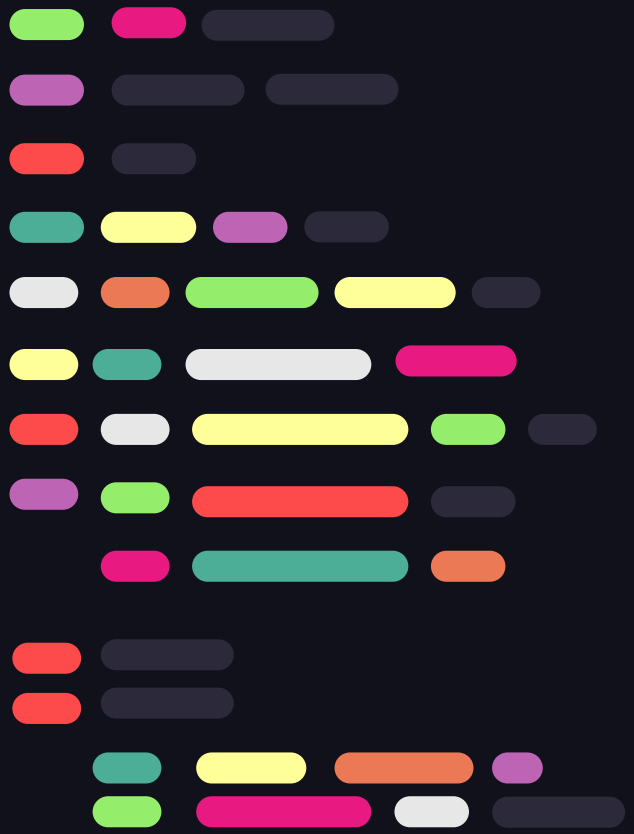
NAME / TYPE	DESCRIPTION
errorPolicy	Specifies how the query handles a response that returns both GraphQL errors and partial results.
onCompleted	A callback function that's called when your query successfully completes with zero errors (or if errorPolicy is ignore and partial data is returned).
skip	If true, the query is not executed. Not available with useLazyQuery.
onError	A callback function that's called when the query encounters one or more errors (unless errorPolicy is ignore).

2.9 useQuery API

Operation options	Networking options	Caching options	Deprecated options
query	pollInterval	fetchPolicy	partialRefetch
variables	notifyOnNetworkStatusChange	nextFetchPolicy	
	context	returnPartialData	
	Ssr / client		

2.10 Result

Operation data	Network info	Helper functions
data	loading	Refetch /fetchMore
Previous Data	networkStatus	startPolling / stopPolling
error	client	subscribeToMore
variables	called	updateQuery



Part 02