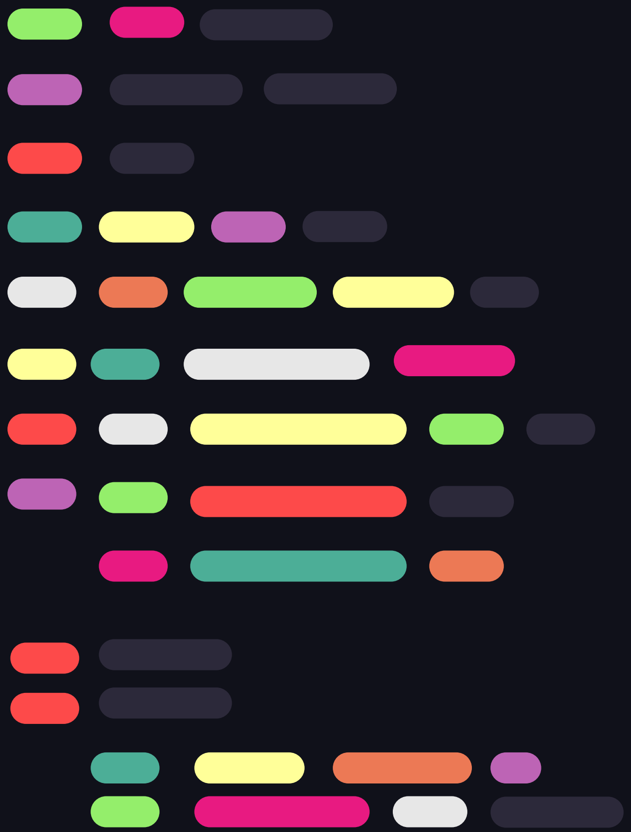
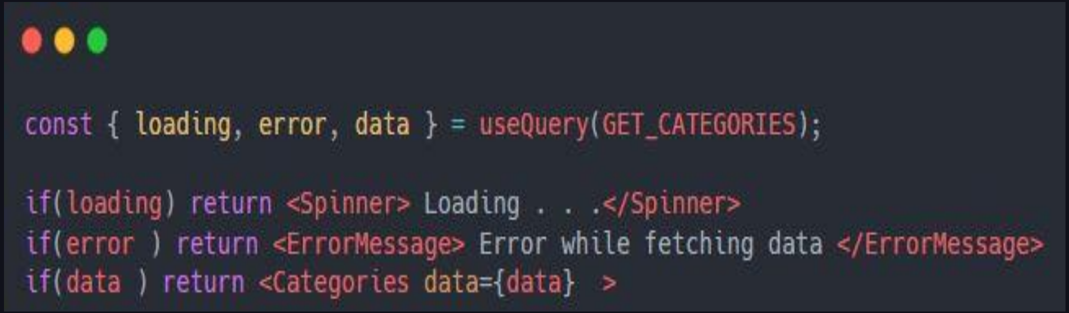
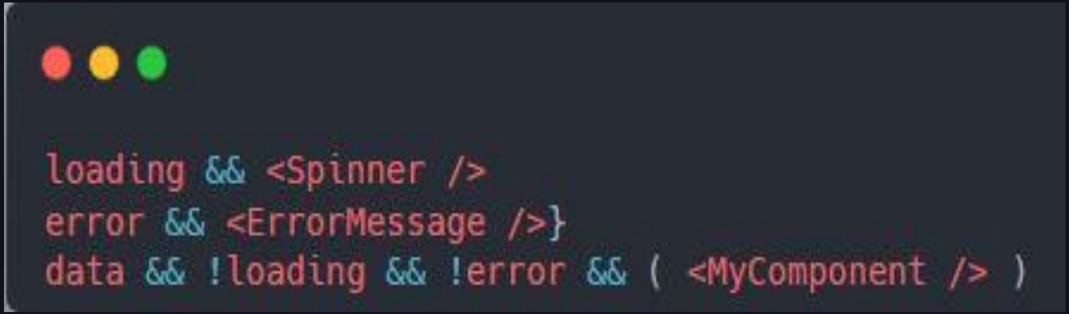


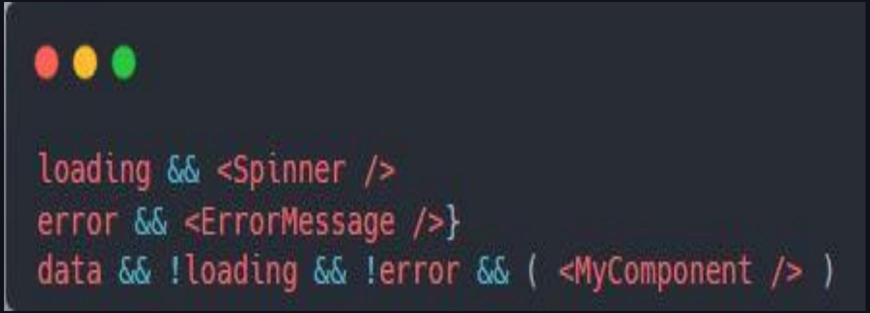
Part
02



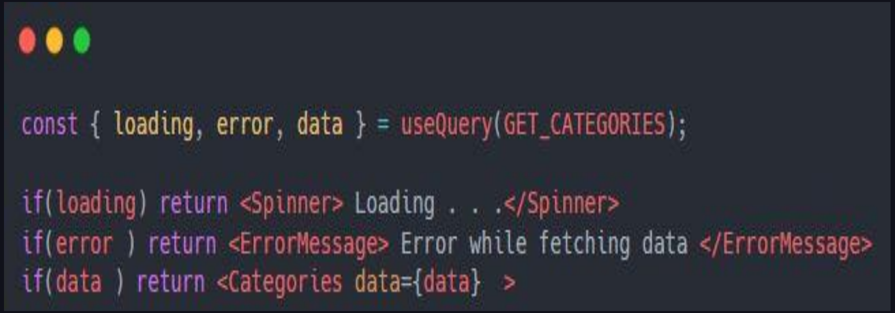
```
const { loading, error, data } = useQuery(GET_CATEGORIES);

if(loading) return <Spinner> Loading . . .</Spinner>
if(error ) return <ErrorMessage> Error while fetching data </ErrorMessage>
if(data ) return <Categories data={data}  >
```



```
loading && <Spinner />
error && <ErrorMessage />}
data && !loading && !error && ( <MyComponent /> )
```



```
loading && <Spinner />
error && <ErrorMessage />}
data && !loading && !error && ( <MyComponent /> )
```



```
const { loading, error, data } = useQuery(GET_CATEGORIES);

if(loading) return <Spinner> Loading . . .</Spinner>
if(error ) return <ErrorMessage> Error while fetching data </ErrorMessage>
if(data ) return <Categories data={data}  >
```

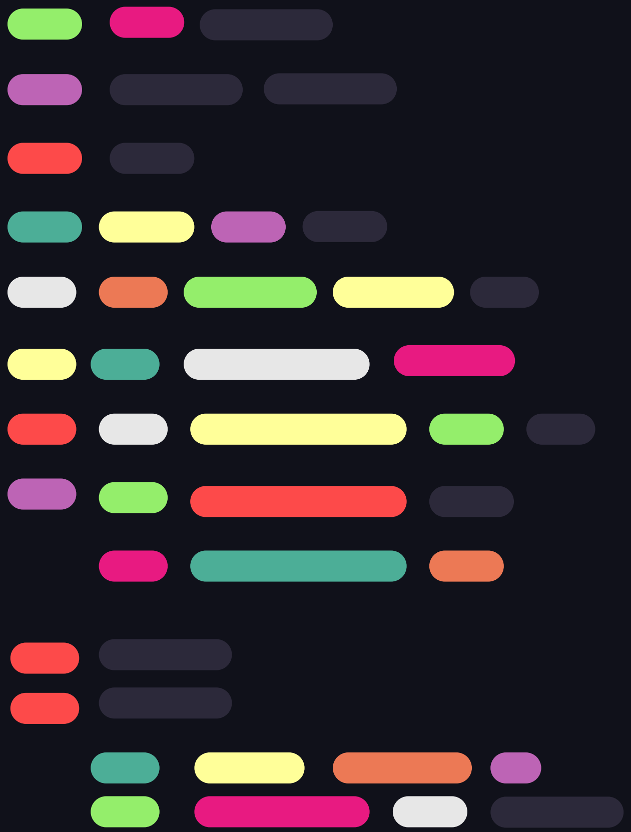
- Ugly Tenaries (BAD DX)
- Boilerplate (BAD DX)
- Confined data loading state (BAD DX & UX)
- Re-fetching data (BAD DX)
- Flashing spinners (BAD UX)

Suspense With Apollo Client

< Presented By Benmoussa Younes />

OmranSoftware - 2023

Table of contents



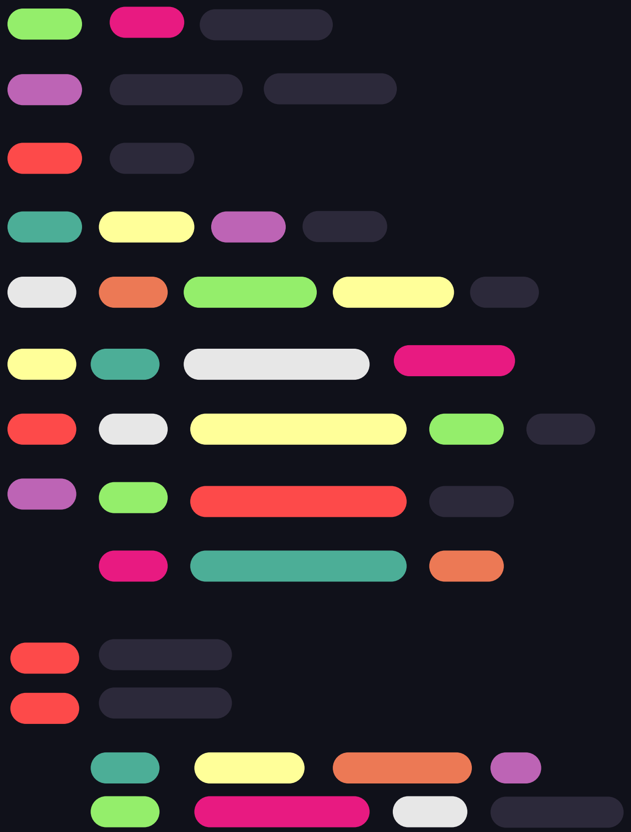
Suspense }

< "Suspense" is generally used to refer to a new way of building React apps using the concurrent rendering engine introduced in React 18 />

Why suspense

Suspense was pitched as an improvement to the developer experience when dealing with asynchronous data fetching

This is a huge deal, because everyone who is building dynamic web applications knows that this is still one of the major pain points.



React 18 }

< What new ? />

React 18 What new ?

Concurrent React

prioritize what component renders, and you can update the part of the component tree that changed

Automatic Batching

Batching is when React groups multiple state updates into a single re-render for better performance.

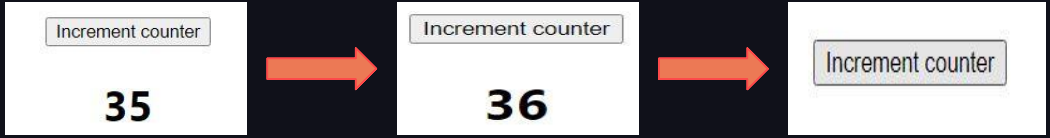
Suspense

An API that can be used to suspense the component execution. It is a way to show a fallback while the component is suspended.

React 18 What new ?

{ Concurrent React

```
const [showCounter, setShowCounter] = useState(false);
const [count, setCount] = useState(0);
const [isPending, startTransition] = useTransition();
//
const onClick = () => {
  startTransition(() => {
    setShowCounter((prev) => !prev );
  });
  setCount((prev) => prev + 1 );
};
```



Automatic Batching

Before

~~Re-render 1 + Re-render 2 = 2 Re-renders~~

```
const onClick = () => {
  setFire(true);
  setEmergency(true);
}
```

Now

1 Re-render }

Suspense

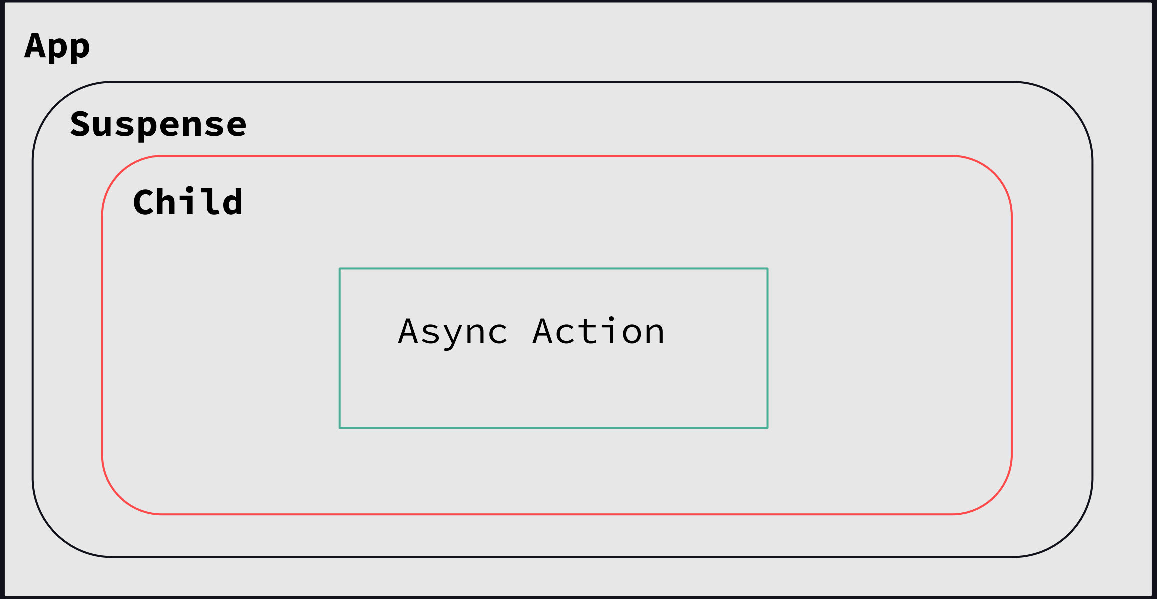
Suspense is a feature for managing asynchronous operations in a React app.
It lets your components communicate to React that they are waiting for some data.

Suspense is not a data fetching library nor is it a way to manage state like Redux.

It simply lets you render a fallback declaratively while a component is waiting for some asynchronous operation (i.e., a network request) to be completed.

Suspense

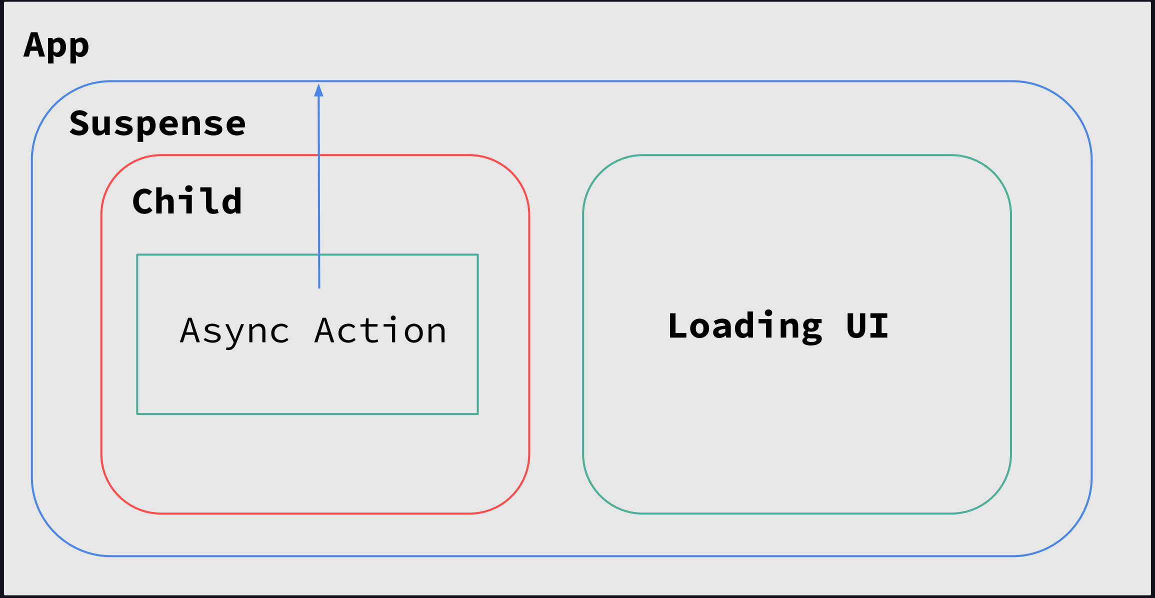
Child component performs some form of asynchronous action



Suspense

Once we perform the asynchronous action(API REQ)

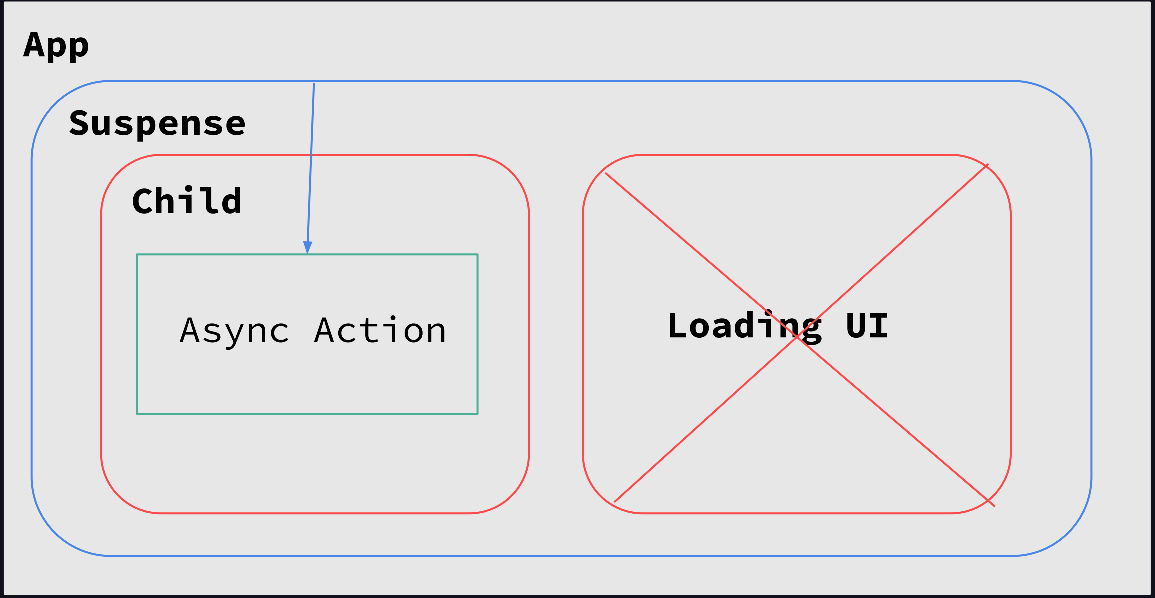
Suspense automatically detects this and shifts to rendering the loading UI



Suspense

Once the data has been returned to us from the API

Suspense detects that the request was completed automatically

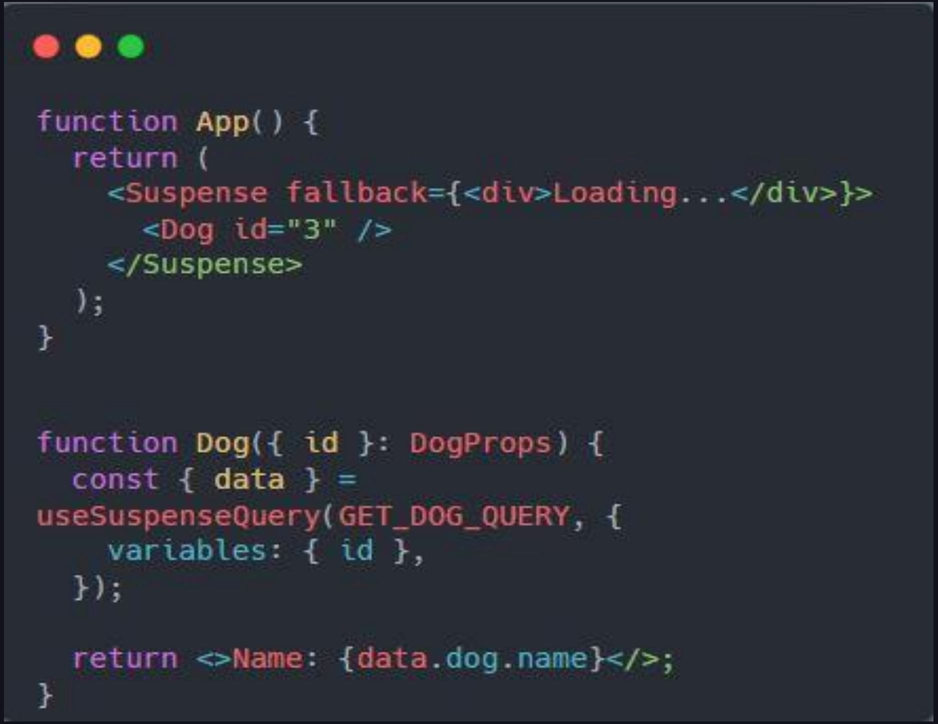


2.8.1 useSuspenseQuery

The useSuspenseQuery hook initiates a network request and causes the component calling it to suspend while the request is made.

You can think of it as a replacement for useQuery that lets you take advantage of React's Suspense features while fetching during render.

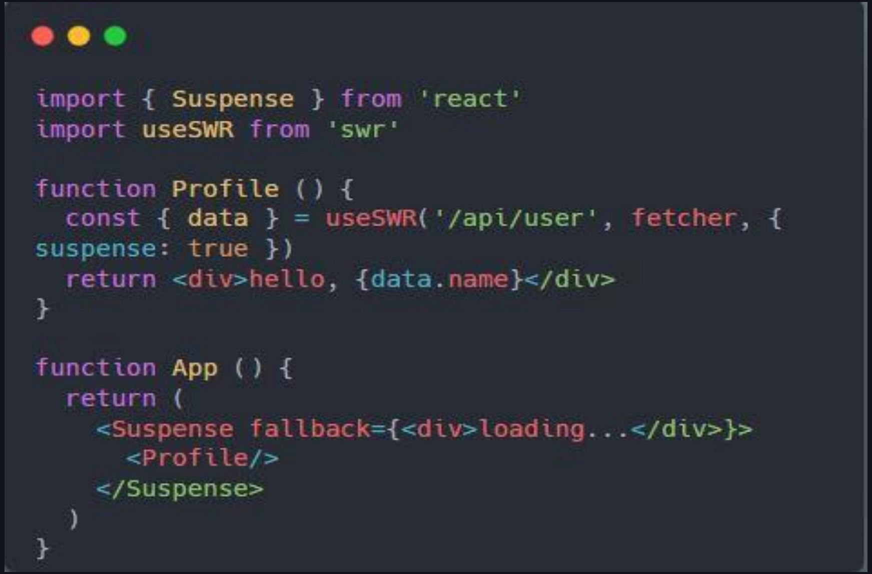
Practical Example 2.8.1



```
function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Dog id="3" />
    </Suspense>
  );
}


function Dog({ id }: DogProps) {
  const { data } =
useSuspenseQuery(GET_DOG_QUERY, {
    variables: { id },
  });

  return <>Name: {data.dog.name}</>;
}
```

Practical Example 2.8.1

Example of fetching with suspense using SWR library

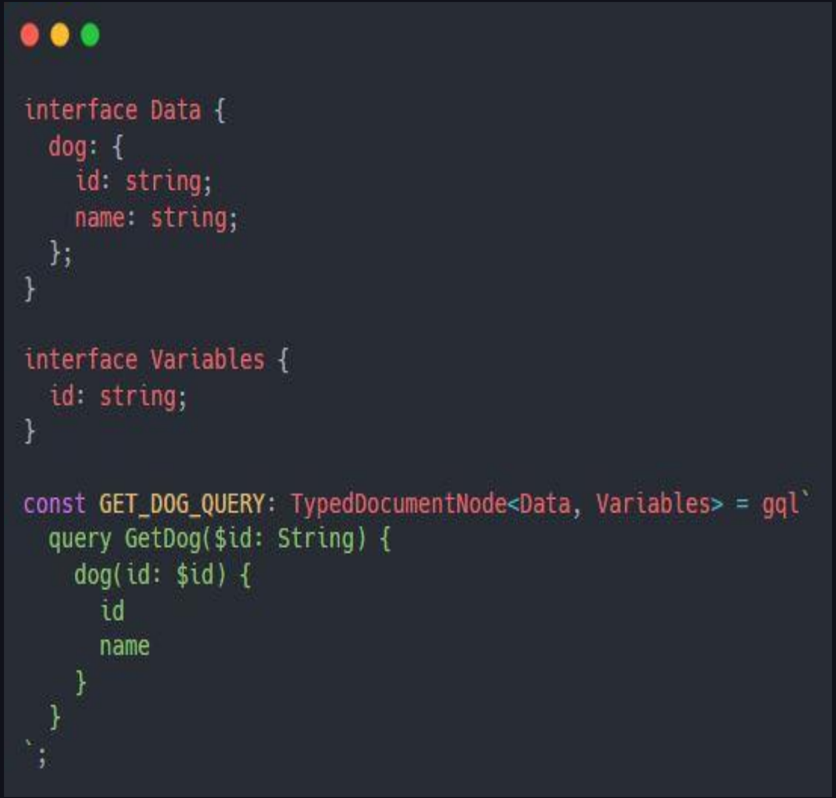


```
import { Suspense } from 'react'
import useSWR from 'swr'

function Profile () {
  const { data } = useSWR('/api/user', fetcher, {
suspense: true })
  return <div>hello, {data.name}</div>
}

function App () {
  return (
    <Suspense fallback={<div>loading...</div>}>
      <Profile/>
    </Suspense>
  )
}
```

2.8.1 useSuspenseQuery

In TypeScript, all APIs that intake DocumentNode can alternatively take TypedDocumentNode<Data, Variables>.

This type enables APIs to infer the data and variable types (instead of making you specify types upon invocation).



```
interface Data {
  dog: {
    id: string;
    name: string;
  };
}

interface Variables {
  id: string;
}

const GET_DOG_QUERY: TypedDocumentNode<Data, Variables> = gql`
  query GetDog($id: String) {
    dog(id: $id) {
      id
      name
    }
  }
`;
```

2.8.2 Changing variables

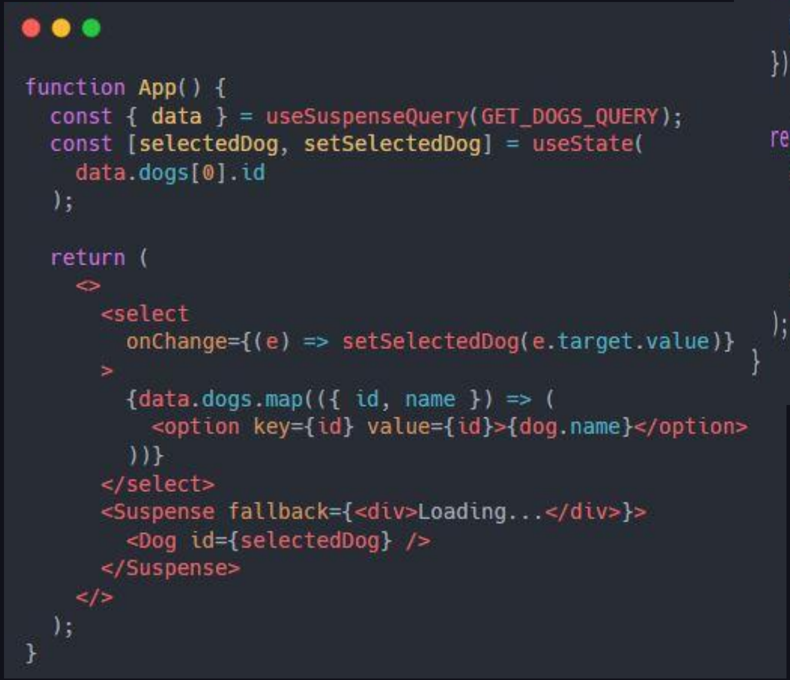
In the previous example, we fetched the record for a single dog by passing a hard-coded id variable to useSuspenseQuery.

Now, let's say we want to fetch the record for a different dog using a dynamic value.

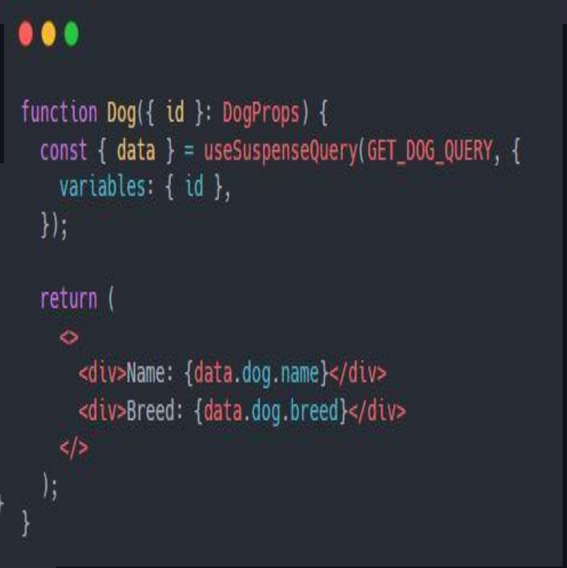
We'll fetch the name and id for our list of dogs, and once the user selects an individual dog, we fetch more details, including their breed.

Practical Example 2.8.2

Let's update our example :



```
function App() {
  const { data } = useSuspenseQuery(GET_DOGS_QUERY);
  const [selectedDog, setSelectedDog] = useState(
    data.dogs[0].id
  );

  return (
    <>
      <select
        onChange={(e) => setSelectedDog(e.target.value)}
      >
        {data.dogs.map(({ id, name }) => (
          <option key={id} value={id}>{dog.name}</option>
        ))}
      </select>
      <Suspense fallback={<div>Loading...</div>}>
        <Dog id={selectedDog} />
      </Suspense>
    </>
  );
}
```



```
function Dog({ id }: DogProps) {
  const { data } = useSuspenseQuery(GET_DOG_QUERY, {
    variables: { id },
  });

  return (
    <>
      <div>Name: {data.dog.name}</div>
      <div>Breed: {data.dog.breed}</div>
    </>
  );
}
```

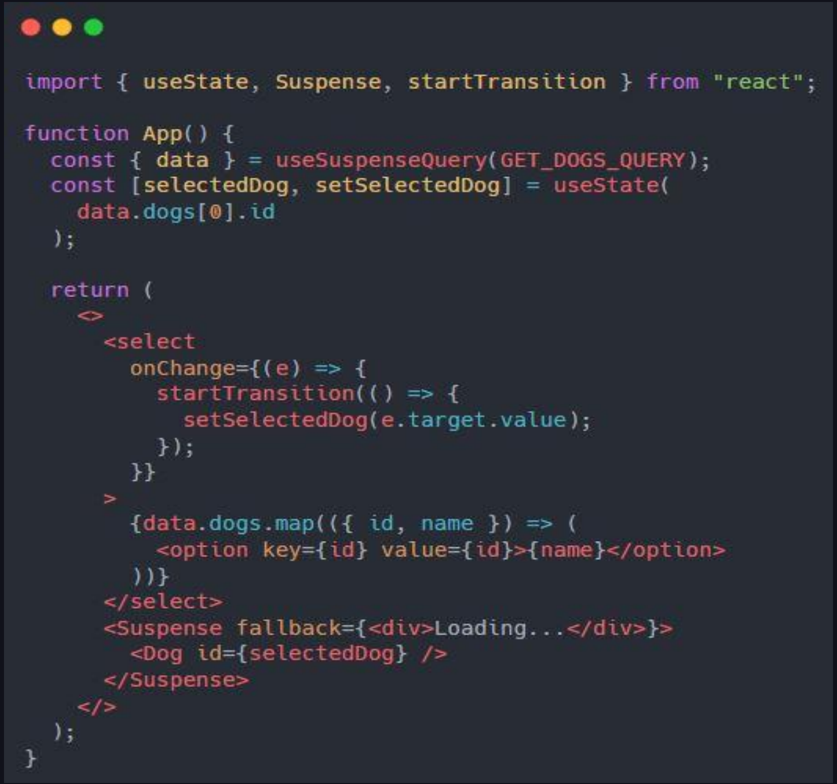
2.8.3 Updating state without suspending

Sometimes we may want to avoid showing a loading UI in response to a pending network request and instead prefer to continue displaying the previous render.

To do this, we can use a transition to mark our update as non-urgent.

This tells React to keep the existing UI in place until the new data has finished loading.

Practical Example 2.8.3



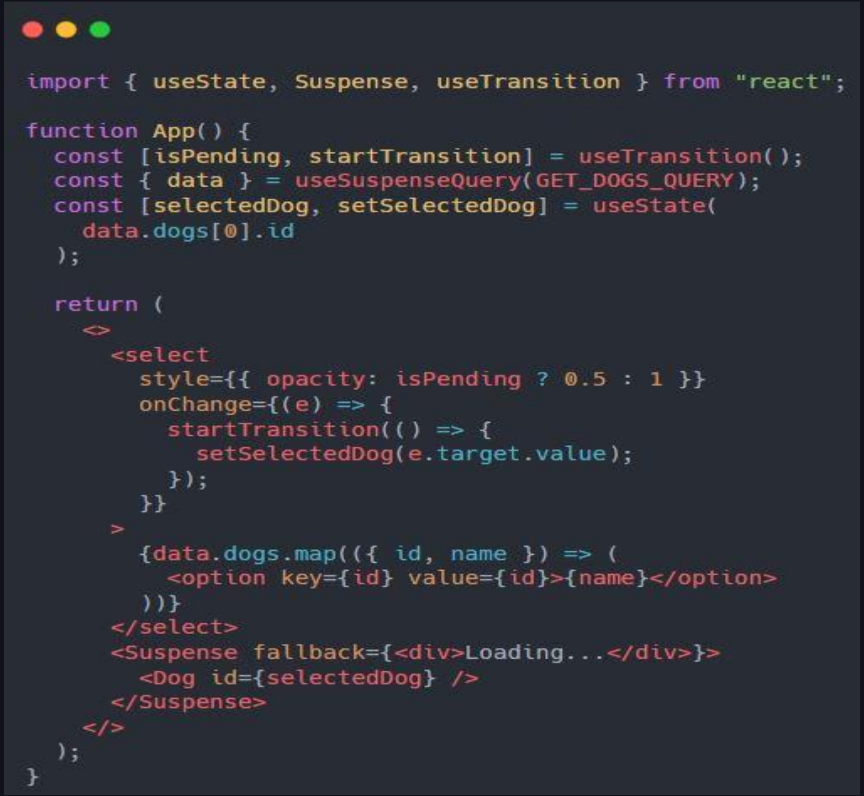
```
import { useState, Suspense, startTransition } from "react";

function App() {
  const { data } = useSuspenseQuery(GET_DOGS_QUERY);
  const [selectedDog, setSelectedDog] = useState(
    data.dogs[0].id
  );

  return (
    <>
      <select
        onChange={(e) => {
          startTransition(() => {
            setSelectedDog(e.target.value);
          });
        }}
      >
        {data.dogs.map(({ id, name }) => (
          <option key={id} value={id}>{name}</option>
        ))}
      </select>
      <Suspense fallback={<div>Loading...</div>}>
        <Dog id={selectedDog} />
      </Suspense>
    </>
  );
}
```

2.8.4 Showing pending UI During a transition

In the previous example, there is no visual indication that a fetch is happening when a new dog is selected.

To provide nice visual feedback, let's update our example to use React's useTransition hook which gives you an isPending boolean value to determine when a transition is happening.

Practical Example 2.8.4



```jsx
import { useState, Suspense, useTransition } from "react";

function App() {
  const [isPending, startTransition] = useTransition();
  const { data } = useSuspenseQuery(GET_DOGS_QUERY);
  const [selectedDog, setSelectedDog] = useState(
    data.dogs[0].id
  );

  return (
    <>
      <select
        style={{ opacity: isPending ? 0.5 : 1 }}
        onChange={(e) => {
          startTransition(() => {
            setSelectedDog(e.target.value);
          });
        }}
      >
        {data.dogs.map(({ id, name }) => (
          <option key={id} value={id}>{name}</option>
        ))}
      </select>
      <Suspense fallback={<div>Loading...</div>}>
        <Dog id={selectedDog} />
      </Suspense>
    </>
  );
}
```

2.8.5 Rendering partial data

When the cache contains partial data, you may prefer to render that data immediately without suspending.

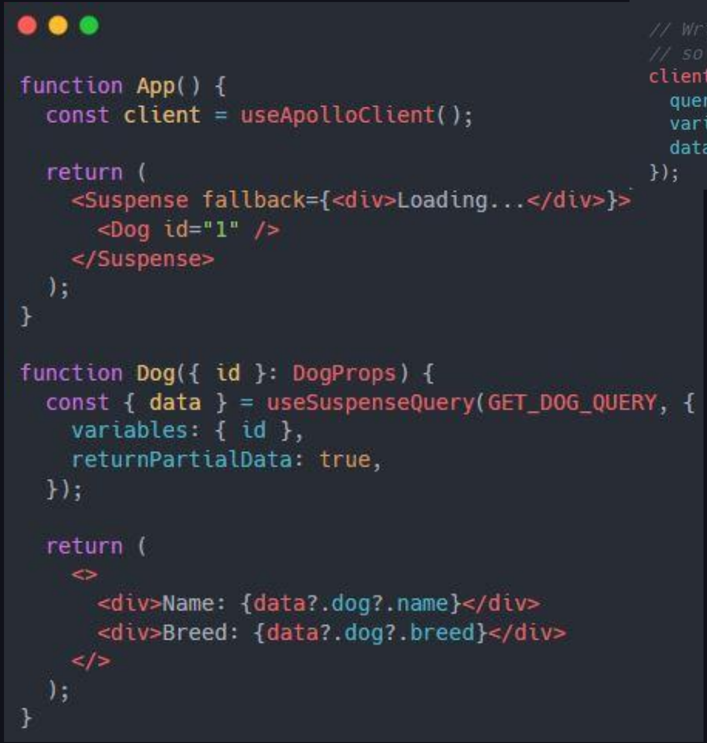
To do this, use the returnPartialData option.

This Option Works with :

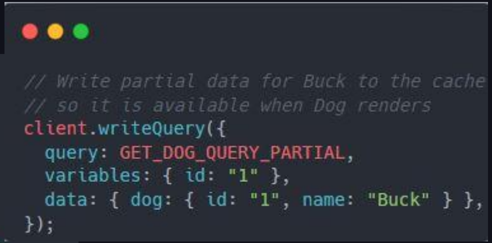
- Cache first (default) or cache-and-network fetch policy

! cache-only is not currently supported by useSuspenseQuery

Practical Example 2.8.5



```
function App() {
  const client = useApolloClient();

  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Dog id="1" />
    </Suspense>
  );
}

function Dog({ id }: DogProps) {
  const { data } = useSuspenseQuery(GET_DOG_QUERY, {
    variables: { id },
    returnPartialData: true,
  });

  return (
    <>
      <div>Name: {data?.dog?.name}</div>
      <div>Breed: {data?.dog?.breed}</div>
    </>
  );
}
```

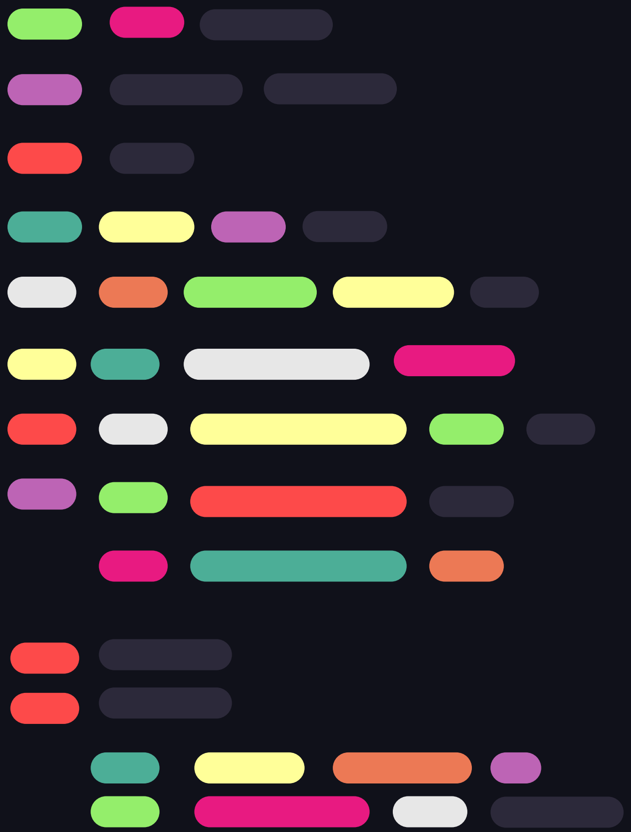


```
// Write partial data for Buck to the cache
// so it is available when Dog renders
client.writeQuery({
  query: GET_DOG_QUERY_PARTIAL,
  variables: { id: "1" },
  data: { dog: { id: "1", name: "Buck" } },
});
```

2.8.5 Rendering partial data

On first render, Buck's name is displayed after the Name label, followed by the Breed label with no value.

Once the missing fields have loaded, useSuspenseQuery triggers a re-render and Buck's breed is displayed.



We forget something no ??

2.8.6 Error handling

By default, both network errors and GraphQL errors are thrown by useSuspenseQuery.

These errors are caught and displayed by the closest error boundary.

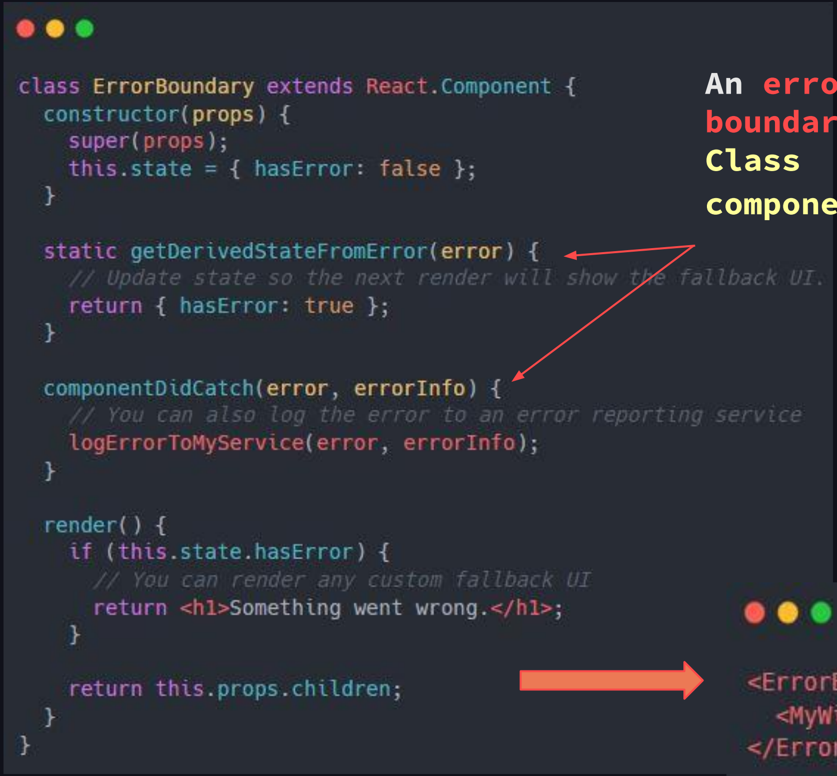
Error boundary ?

2.8.6 Error Boundary

A JavaScript error in a part of the UI shouldn't break the whole app.

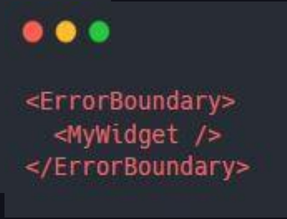
To solve this problem for React users, React 16 introduces a new concept of an "error boundary".

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.



```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

An error boundary Class component

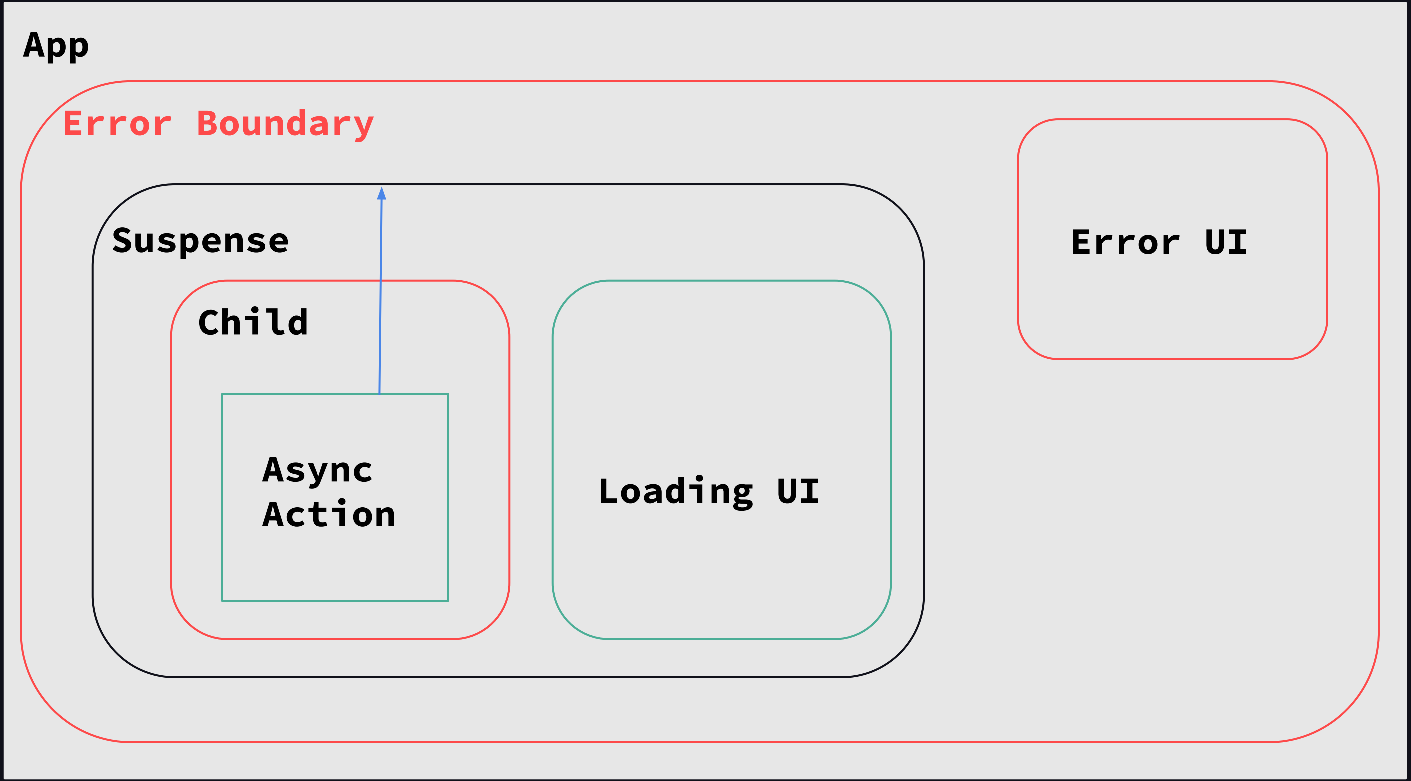


```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

2.8.6 Error Boundary

The asynchronous action is performed, and the loading UI

Error UI is not displayed because no error has occurred

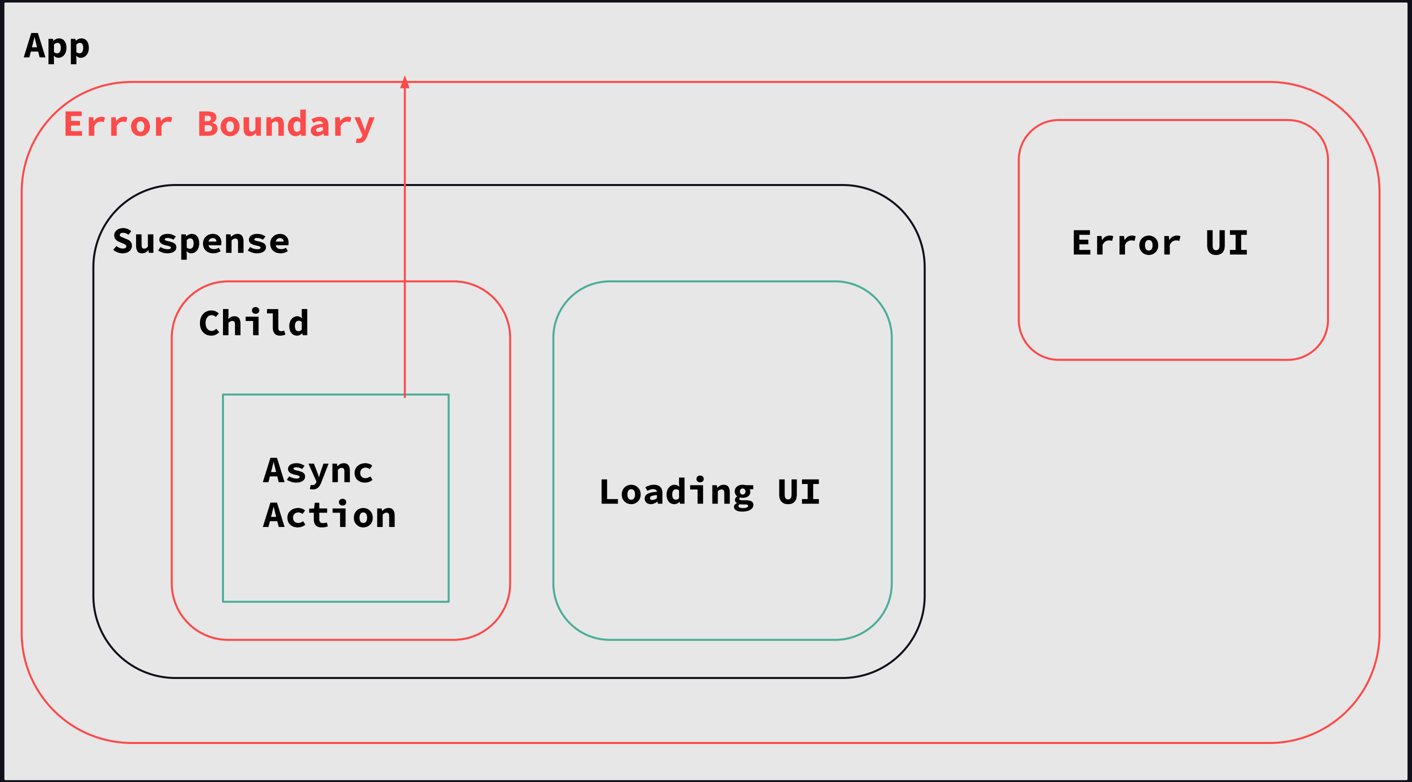


2.8.6 Error Boundary

if an error does occur,

remove the loading UI,

automatically display the error UI

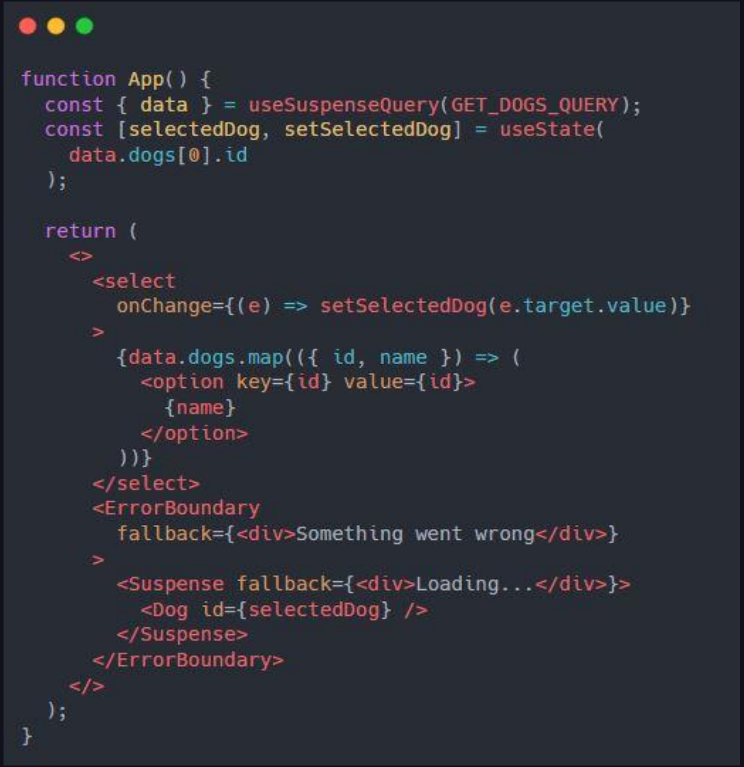


2.8.6 Error handling

When the GET_DOG_QUERY inside of the Dog component returns a GraphQL error or a network error, useSuspenseQuery throws the error and the nearest error boundary renders its fallback component.

In some cases, you may want to render partial data alongside an error.

To do this, set the errorPolicy option to all.



```
function App() {
  const { data } = useSuspenseQuery(GET_DOGS_QUERY);
  const [selectedDog, setSelectedDog] = useState(
    data.dogs[0].id
  );

  return (
    <>
      <select
        onChange={(e) => setSelectedDog(e.target.value)}
      >
        {data.dogs.map(({ id, name }) => (
          <option key={id} value={id}>
            {name}
          </option>
        ))}
      </select>
      <ErrorBoundary
        fallback={<div>Something went wrong</div>}
      >
        <Suspense fallback={<div>Loading...</div>}>
          <Dog id={selectedDog} />
        </Suspense>
      </ErrorBoundary>
    </>
  );
}
```

*

2.8.7 Avoiding request waterfalls

Since useSuspenseQuery suspends while data is being fetched, a tree of components that all use useSuspenseQuery can cause a "waterfall",

where each call to useSuspenseQuery depends on the previous to complete before it can start fetching.

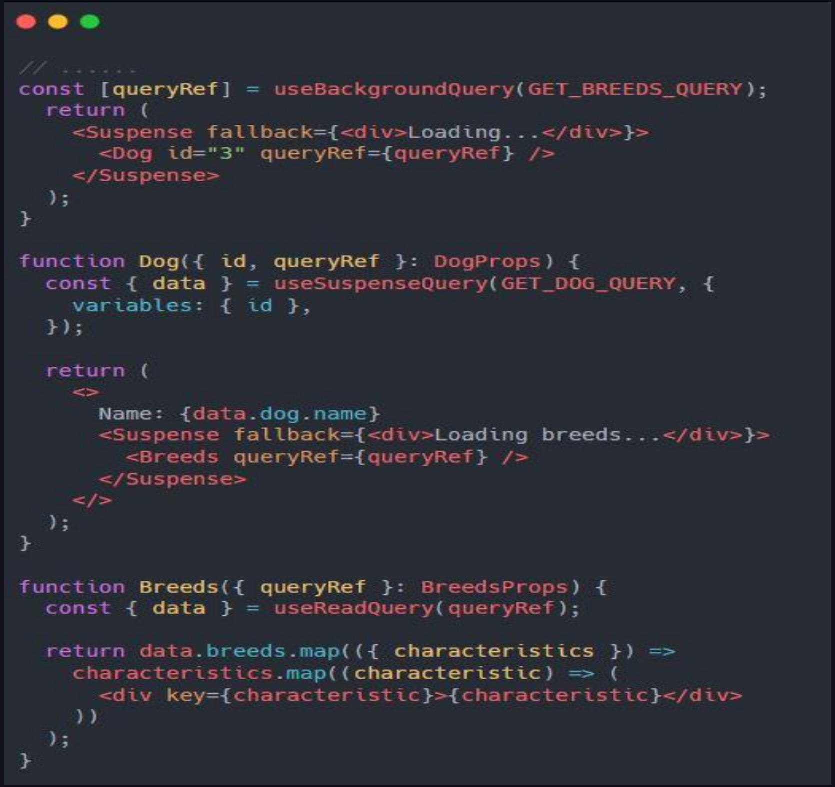
This can be avoided by fetching with useBackgroundQuery and reading the data with useReadQuery.

2.8.7 Avoiding request waterfalls

useBackgroundQuery initiates a request for data in a parent component and returns a queryRef which is passed to useReadQuery to read the data in a child component.

When the child component renders before the data has finished loading, the child component suspends.

Let's update our example to utilize useBackgroundQuery



```
// ......
const [queryRef] = useBackgroundQuery(GET_BREEDS_QUERY);
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Dog id="3" queryRef={queryRef} />
    </Suspense>
  );
}

function Dog({ id, queryRef }: DogProps) {
  const { data } = useSuspenseQuery(GET_DOG_QUERY, {
    variables: { id },
  });

  return (
    <>
      Name: {data.dog.name}
      <Suspense fallback={<div>Loading breeds...</div>}>
        <Breeds queryRef={queryRef} />
      </Suspense>
    </>
  );
}

function Breeds({ queryRef }: BreedsProps) {
  const { data } = useReadQuery(queryRef);

  return data.breeds.map(({ characteristics }) =>
    characteristics.map((characteristic) => (
      <div key={characteristic}>{characteristic}</div>
    ))
  );
}
```

2.8.7 Avoiding request waterfalls

! A note about performance

The useBackgroundQuery hook used in a parent component is responsible for kicking off fetches, but doesn't deal with reading or rendering data.

This is delegated to the useReadQuery hook used in a child component.

This separation of concerns provides a nice performance benefit because cache updates are observed by **useReadQuery** and re-render only the child component.

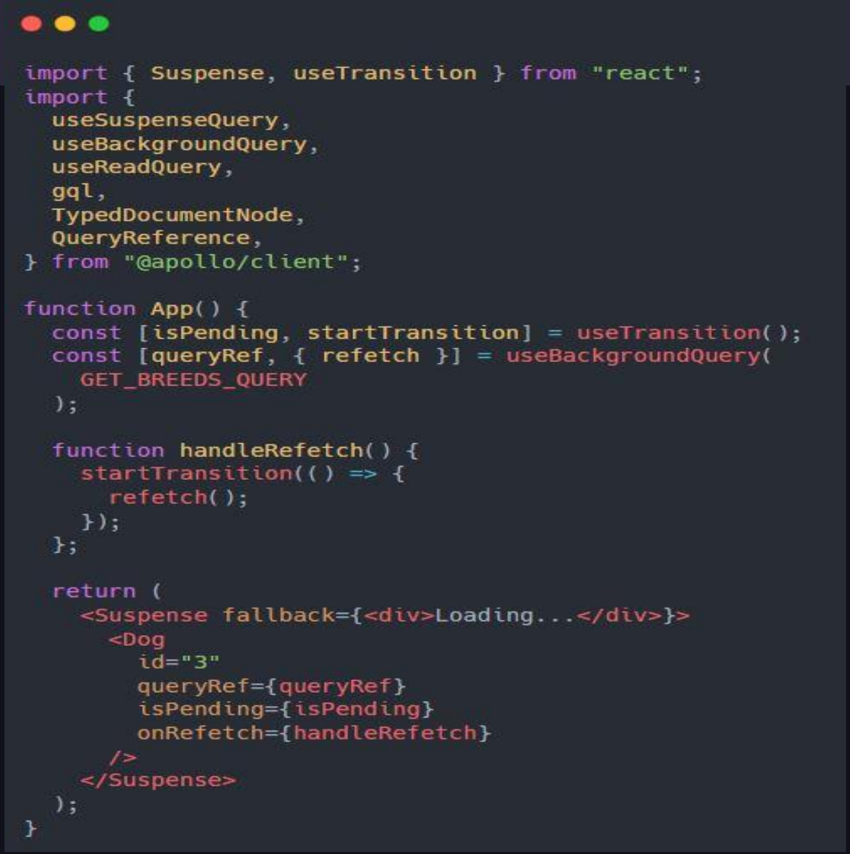
2.8.8 Refetching and pagination

Apollo's Suspense data fetching hooks return functions for refetching query data via the refetch function, and fetching additional pages of data via the fetchMore function.

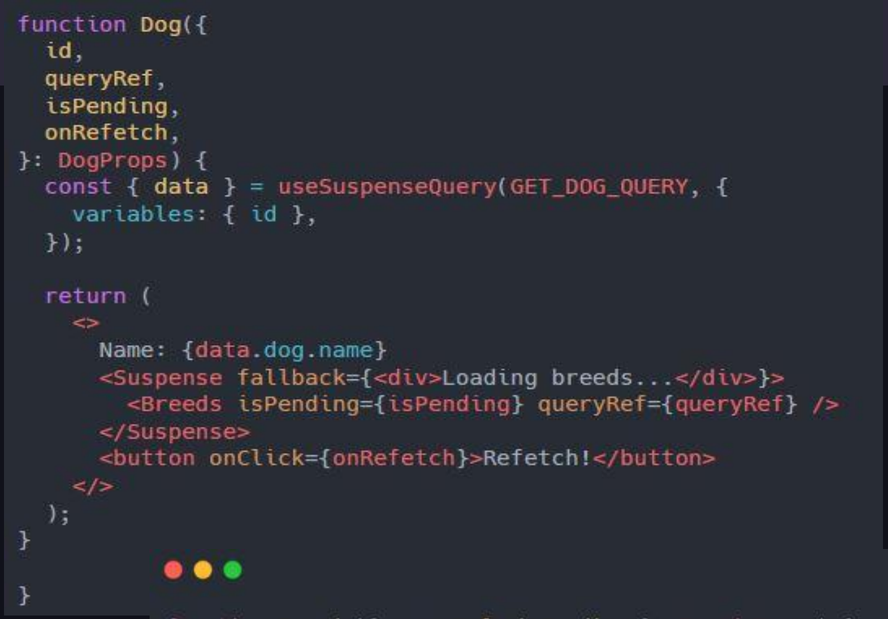
Let's update our example by adding the ability to refetch breeds.

We destructure the refetch function from the second item in the tuple returned from useBackgroundQuery.

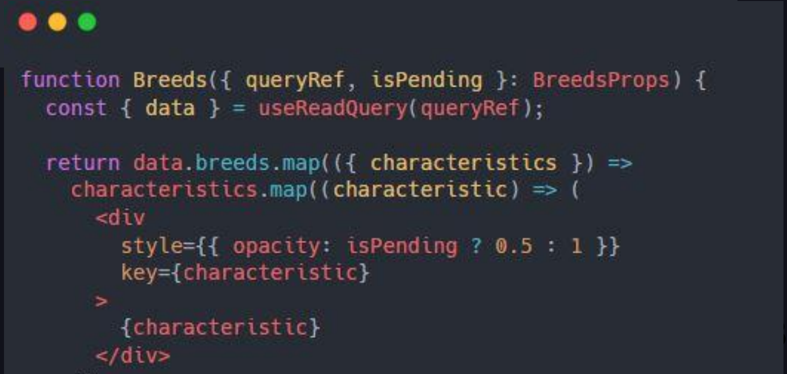
2.8.8 Refetching and pagination



```tsx
import { Suspense, useTransition } from "react";
import {
  useSuspenseQuery,
  useBackgroundQuery,
  useReadQuery,
  gql,
  TypedDocumentNode,
  QueryReference,
} from "@apollo/client";

function App() {
  const [isPending, startTransition] = useTransition();
  const [queryRef, { refetch }] = useBackgroundQuery(
    GET_BREEDS_QUERY
  );

  function handleRefetch() {
    startTransition(() => {
      refetch();
    });
  };

  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Dog
        id="3"
        queryRef={queryRef}
        isPending={isPending}
        onRefetch={handleRefetch}
      />
    </Suspense>
  );
}
```



```tsx
function Dog({
  id,
  queryRef,
  isPending,
  onRefetch,
}: DogProps) {
  const { data } = useSuspenseQuery(GET_DOG_QUERY, {
    variables: { id },
  });

  return (
    <>
      Name: {data.dog.name}
      <Suspense fallback={<div>Loading breeds...</div>}>
        <Breeds isPending={isPending} queryRef={queryRef} />
      </Suspense>
      <button onClick={onRefetch}>Refetch!</button>
    </>
  );
}

}
```



```tsx
function Breeds({ queryRef, isPending }: BreedsProps) {
  const { data } = useReadQuery(queryRef);

  return data.breeds.map(({ characteristics }) =>
    characteristics.map((characteristic) => (
      <div
        style={{ opacity: isPending ? 0.5 : 1 }}
        key={characteristic}
      >
        {characteristic}
      </div>
```

2.8.9 Distinguishing between queries with queryKey

Apollo Client uses the combination of query and variables to uniquely identify each query when using Apollo's Suspense data fetching hooks.

If your application renders multiple components that use the same query and variables, this may present a problem:

! the queries made by multiple hooks share the same identity causing them to suspend at the same time, regardless of which component initiates or re-initiates a network request.

2.8.9 Distinguishing between queries with queryKey

Apollo Client uses the combination of query and variables to uniquely identify each query when using Apollo's Suspense data fetching hooks.

If your application renders multiple components that use the same query and variables, this may present a problem:

! the queries made by multiple hooks share the same identity causing them to suspend at the same time, regardless of which component initiates or re-initiates a network request.

2.8.9 Distinguishing between queries with queryKey

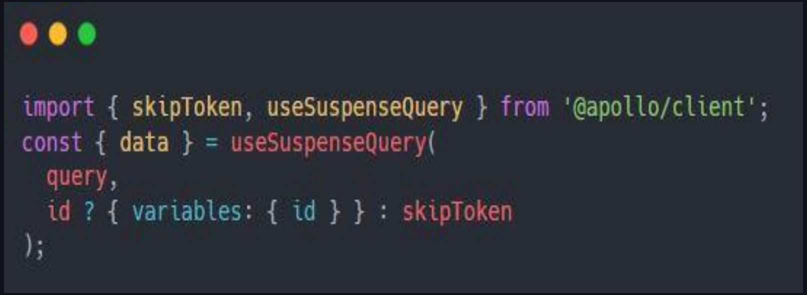
You can prevent this with queryKey option to ensure each hook has a unique identity.

When queryKey is provided, Apollo Client uses it as part of the hook's identity in addition to its query and variables.

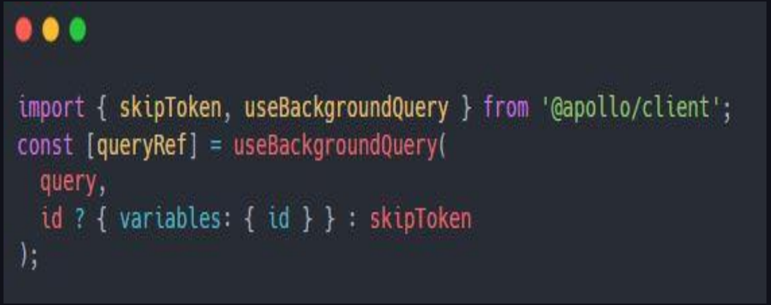
2.9 Skipping suspense hooks

While useSuspenseQuery and useBackgroundQuery both have a skip option, that option is only present to ease migration from useQuery with as few code changes as possible.

It should not be used in the long term. Instead, you should use skipToken



```
import { skipToken, useSuspenseQuery } from '@apollo/client';
const { data } = useSuspenseQuery(
  query,
  id ? { variables: { id } } : skipToken
);
```



```
import { skipToken, useBackgroundQuery } from '@apollo/client';
const [queryRef] = useBackgroundQuery(
  query,
  id ? { variables: { id } } : skipToken
);
```

2.9.1 React Server Components (RSC)

In Next.js v13, Next.js's new App Router brought the React community the first framework with full support for React Server Components (RSC) and Streaming SSR, integrating Suspense as a first-class concept from your application's routing layer all the way down.

2.9.1 React Server Components (RSC)

Error handling

In a purely client-rendered app, errors thrown in components are always caught and displayed by the closest error boundary.

Errors thrown on the server when using one of the streaming server rendering APIs are treated differently.

2.9.2 useSuspenseQuery API

{

Operation options	Networking options	Caching options	Result
variables	context	fetchPolicy	data
errorPolicy	canonizeResults	returnPartialData	error
	client	refetchWritePolicy	networkStatus
	queryKey	skip (deprecated)	

}

2.9.2 useSuspenseQuery API

{

Operation options	Networking options	Caching options	Result
variables	context	fetchPolicy	data
errorPolicy	canonizeResults	returnPartialData	error
	client	refetchWritePolicy	networkStatus
	queryKey	skip (deprecated)	

}

2.9.2 useSuspenseQuery API

Helper Fn	DESCRIPTION
refetch	A function that enables you to re-execute the query, optionally passing in new variables.
fetchMore	A function that helps you fetch the next set of results for a paginated list field.
subscribeToMore	A function that enables you to execute a subscription, usually to subscribe to specific fields that were included in the query.

! Calling this functions will cause the component to re-suspend, unless the call site is wrapped in startTransition .

2.9.3 useBackgroundQuery API

{

Operation options	Networking options	Caching options	Result
variables	context	fetchPolicy	data
errorPolicy	canonizeResults	returnPartialData	error
	client	refetchWritePolicy	networkStatus
		skip (deprecated)	

}

2.9.3 useBackgroundQuery API

{

Helper Fn	DESCRIPTION
refetch	A function that enables you to re-execute the query, optionally passing in new variables.
fetchMore	A function that helps you fetch the next set of results for a paginated list field.

! Calling this functions will cause the component to re-suspend, unless the call site is wrapped in startTransition .

}

2.9.4 useReadQuery API

{

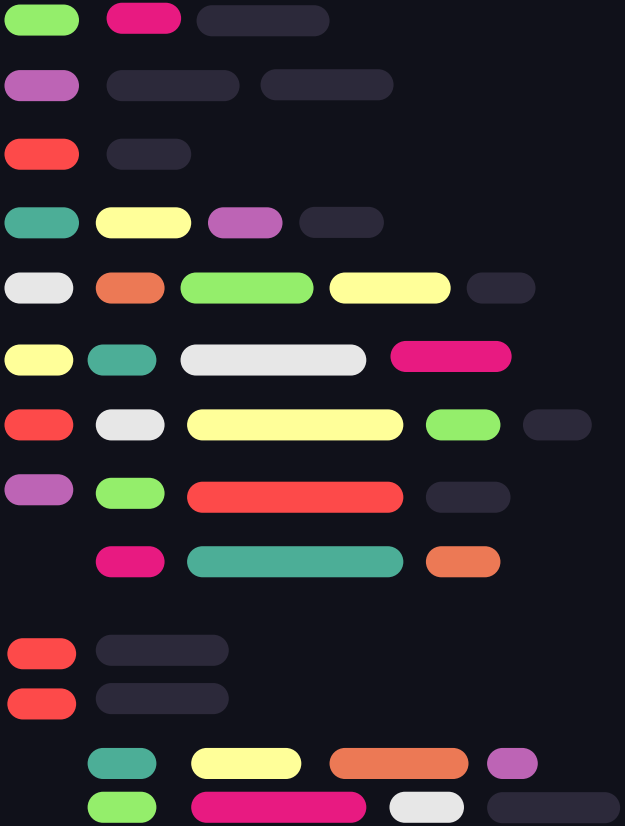
Operation options	Networking options	Caching options	Result
variables	context	fetchPolicy	data
errorPolicy	canonizeResults	returnPartialData	error
	client	refetchWritePolicy	networkStatus
		skip (deprecated)	

}

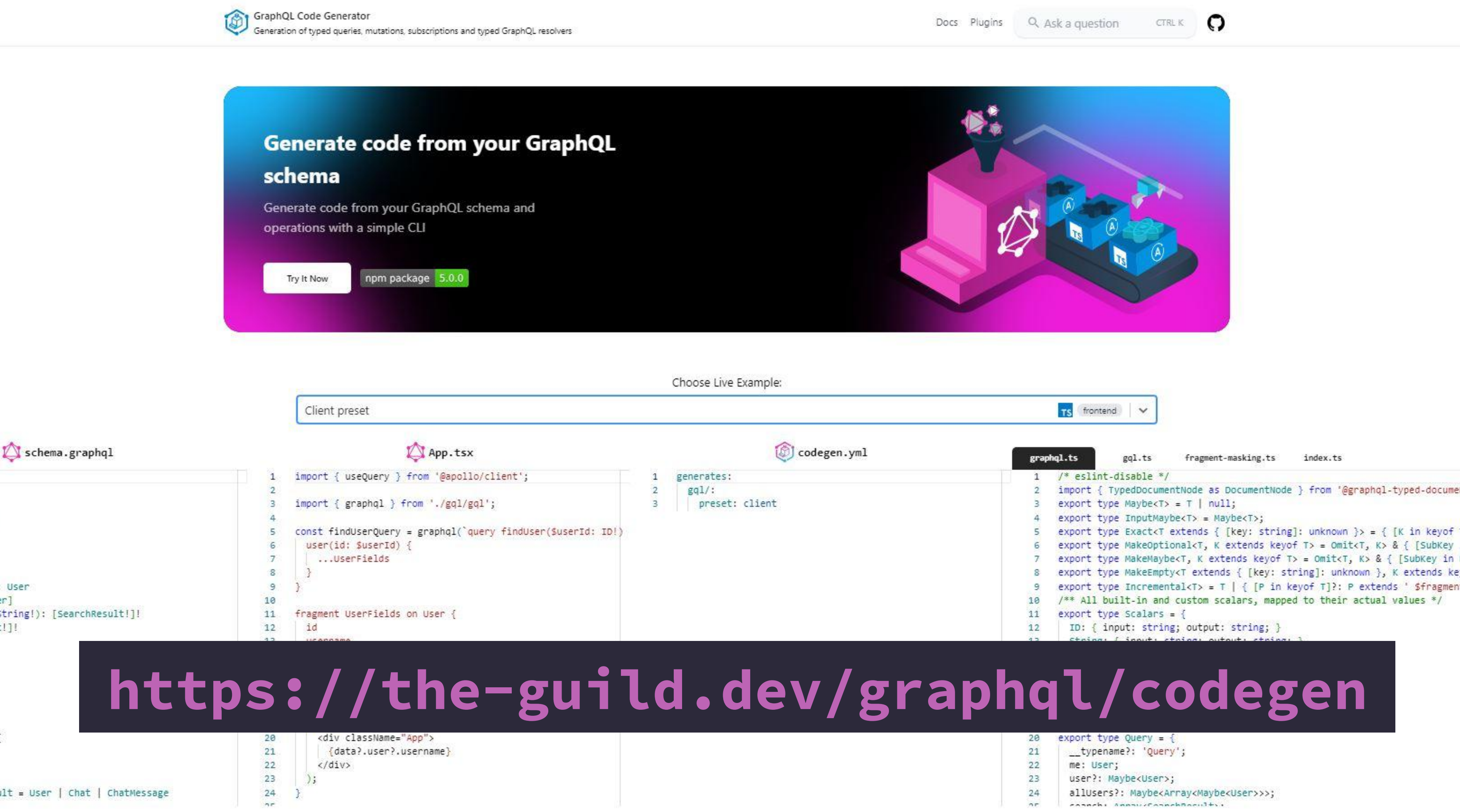
2.9.5 skipToken

While not a hook by itself, skipToken is designed to be used with useSuspenseQuery and useBackgroundQuery.

If a skipToken is passed into one of those hooks instead of the options object, that hook will not cause any requests or suspenseful behavior, while keeping the last data available.



Bonus Content


GraphQL Code Generator
Generation of typed queries, mutations, subscriptions and typed GraphQL resolvers

Generate code from your GraphQL schema

Generate code from your GraphQL schema and operations with a simple CLI

Choose Live Example:

Client preset

schema.graphql

```
: User
er]
String!): [SearchResult!]!
t!]!
```

```
lt = User | Chat | ChatMessage
```

App.tsx

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

import { graphql } from './gql/gql';

const findUserQuery = graphql(`query findUser($userId: ID!)
  user(id: $userId) {
    ...UserFields
  }
}

fragment UserFields on User {
  id
```

```
    <div className="App">
      {data?.user?.username}
    </div>
  );
}
```

codegen.yml

```
generates:
  gql/:
    preset: client
```

graphql.ts | gql.ts | fragment-masking.ts | index.ts

```
/* eslint-disable */
import { TypedDocumentNode as DocumentNode } from '@graphql-typed-docume
export type Maybe<T> = T | null;
export type InputMaybe<T> = Maybe<T>;
export type Exact<T extends { [key: string]: unknown }> = { [K in keyof
export type MakeOptional<T, K extends keyof T> = Omit<T, K> & { [SubKey
export type MakeMaybe<T, K extends keyof T> = Omit<T, K> & { [SubKey in
export type MakeEmpty<T extends { [key: string]: unknown }, K extends ke
export type Incremental<T> = T | { [P in keyof T]?: P extends ' $fragmen
/** All built-in and custom scalars, mapped to their actual values */
export type Scalars = {
  ID: { input: string; output: string; }
```

```
export type Query = {
  __typename?: 'Query';
  me: User;
  user?: Maybe<User>;
  allUsers?: Maybe<Array<Maybe<User>>>;
```

https://the-guild.dev/graphql/codegen

Alternative resources

- Apollo client suspense