Local State Management with Reactive Variables

In this article, we'll learn how to get started with local state management using Cache Policies and Reactive Variables on TodoApp.

Cache policies

Cache policies are a way to customize reads and writes to the cache.

They give us the control to model types and fields that **might not exist as a part of your data graph**, but *do* exist on the client-side. That's exactly what we take advantage of for *local state management*.

For starters, here's how to initialize a default value for a `text` field on the `Todo` type.

```
const cache = new InMemoryCache({
  typePolicies: {
    Todo: { // Every todo type
      fields: {
        text: { // A field on the todo type
          read (value = "SOME TEXT") { // when it's read
            return value;
          }
        },
      },
    },
  },
});
```

By providing a `read` function for any field you want to configure, we're given the currently cached value of the field, and whatever we return is what's going to be the new value.

It turns out that we can do a lot of things with this API. We can implement filters, handle pagination, and configure local state using Reactive Variables.
Cache policies are half of the local state management story in AC3. Reactive variables are the other part.

Reactive variables

Reactive variables are *containers* for variables that we would like to enable cache reactivity for. Using the small API, we can either:

- set the value by passing in an argument — `var(newValue)`
- get the value by invoking the reactive variable — `const currentValue = var()`

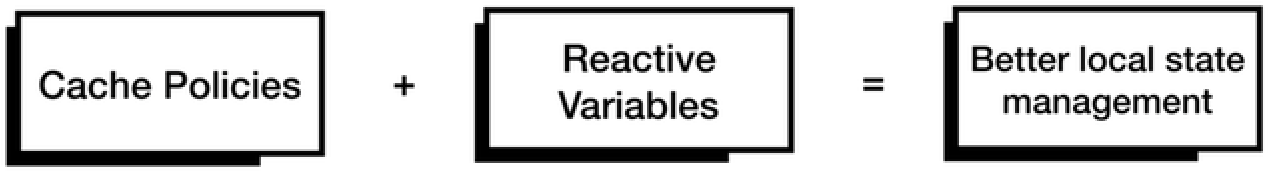
Here's how it works in practice.

```
import { makeVar } from "@apollo/client";

// Create Reactive variable
export const todosVar = makeVar<Todos>();

// Set the value
todosVar([]);

// Get the value
const currentTodosValue = todosVar();
```

Cache *reactivity* means that when the value of reactive variable changes, it **notifies any queries in components that were subscribed to the value of the change**.

Using Cache Policies and Reactive Variables, we can:

- Query local state variables the same way we query for data that exists remotely
- Trigger updates using simple functions



Setting up a Reactive Variable

Let's define an `interface` or a `type` to represent the shape of the variable you want to create.

In this case, we're modeling `Todos` as an array of `Todo` objects.

```
// models/todos.tsx

export interface Todo {
  text: string;
  completed: boolean;
  id: number
}

export type Todos = Todo[];
```

Next, where we've configured our cache, use the `makeVar` method to create a Reactive Variable, optionally passing in an initial value.

```
// cache.tsx

import { InMemoryCache, makeVar } from "@apollo/client";
import { Todos } from './models/todos'

export const cache: InMemoryCache = new InMemoryCache({});

// Create the initial value
const todosInitialValue: Todos = [
  {
    id: 0,
    completed: false,
    text: "Use Apollo Client 3"
  }
]

// Create the todos var and initialize it with the initial value
export const todosVar = makeVar<Todos>(
  todosInitialValue
);
```

Now it's time to connect the Reactive Variable to a cache policy. In the config for the cache object, define a cache policy and connect the value of a `todos` field to the `todosVar` reactive variable we just created.

```
// cache.tsx
...
export const cache: InMemoryCache = new InMemoryCache({
  typePolicies: {
    Query: {
      fields: {
        todos: {
          read () {
            return todosVar();
          }
        }
      }
    }
  }
});
```

That's it! We've just configured reactivity for this fully client-side local variable.

By defining `todos` as a field on the `Query` type, writing a query that asks for `todos` uses the current value of the reactive variable we just set up here.

```
// cache.tsx

import { InMemoryCache, makeVar } from "@apollo/client";
import { Todos } from './models/todos'

const cache = new InMemoryCache({
  typePolicies: {
    Query: {
      fields: {
        todos: {
          read () {
            return todosVar();
          }
        },
      },
    },
  },
});

// Create the initial value
const todosInitialValue: Todos = [
  {
    id: 0,
    completed: false,
    text: "Use Apollo Client 3"
  }
]

// Create the todos var and initialize it with the initial value
export const todosVar = makeVar<Todos>(todosInitialValue)
```

Subscribing to the value of a Reactive Variable

Let’s demonstrate how to get the current value of a reactive variable. First, write a query to fetch the data we want.

```
// operations/queries/GetAllTodos.tsx

import { gql } from "@apollo/client";

export const GET_ALL_TODOS = gql`
  query GetAllTodos {
    todos @client {
      id
      text
      completed
    }
  }
`
```

Apollo Client distinguishes between data that we want to fetch remotely and data we want to fetch from the client through the use of the `@client` directive. By placing it alongside the `todos` keyword, we're telling Apollo Client to attempt to resolve everything nested within the `todos` type from the Apollo Client cache itself instead of trying to resolve it remotely.

To execute this query, use the `useQuery` hook like we would normally.

```
//Container Component
import React from 'react'
import Main from '../components/Main'
import { useQuery } from '@apollo/client'
import { GET_ALL_TODOS } from '../operations/queries/getAllTodos'

export default function MainContainer () {
  const { data } = useQuery(GET_ALL_TODOS);
  const todos: Todos = data.todos;
        ...
  return (
    <Main
      todosCount={todos.length}
      completedCount={todos.filter(t => t.completed).length}
    />
  );
}
```

That's all there is to it!

Updating a Reactive Variable

Updates to reactive variables can be done by importing the variable and invoking it with a new value.

Here's a trivial example of **deleting a todo**.

```
//Container Component
import React from 'react'
import MainSection from '../components/MainSection'
import { todosVar } from '../cache'
import { Todo } from '../models'

export default function Main () {
  const todos = todosVar();
        ...
  return (
    <MainSection
      todosCount={todos.length}
      completedCount={todos.filter(t => t.completed).length}
      actions={{
        // Delete todo
        deleteTodo: (id: number) => todosVar(
                                        todosVar().filter((todo: Todo))
=> todo.id !== id
                                );
      }}
    />
  );
};
```

This works, but remember that we should keep logic that determines how state changes outside of both presentation and container components?

How things change belongs to the model layer, more specifically, the interaction layer. This could be a simple function, or a custom React hook that contains all the operations and client-side only models needed for `todos`. Let’s move this *delete logic* there instead.

```
// hooks/useTodos.tsx

import { Todo, Todos } from "../../../models/Todos";
import { ReactiveVar } from "@apollo/client";
import { todosVar } from '../cache'

function todoMutations (todosVar: ReactiveVar<Todos>) {

  ... // Other todos operations
  const editTodo = (id: number, text: string) => {
    const allTodos = todosVar();
    const todosWithEditedTodo = allTodos
      .map((todo: Todo) => todo.id === id ? { ...todo, text } : todo);

    todosVar(todosWithEditedTodo);
  }

  const deleteTodo = (id: number) => {
    const allTodos = todosVar();
    const filteredTodos = allTodos.filter((todo: Todo) => todo.id !==
id);
    todosVar(filteredTodos);
  }

  return { editTodo, deleteTodo, ... }
}

export const useTodos = () => todoMutations(todosVar)
```

And then we can import the custom hook in our container component and pass it to the presentational components.

```
//Container Component
import React from 'react'
import MainSection from '../components/MainSection'
import { useQuery } from '@apollo/client'
import { Todo } from '../models'
import { useTodos } from '../hooks/useTodos.tsx'

export default function Main () {
  const { data: todos } = useQuery(GET_ALL_TODOS);
  const { editTodo, deleteTodo } = useTodos();
        ...
  return (
    <MainSection
      todosCount={todos.length}
      completedCount={todos.filter(t => t.completed).length}
      actions={{
        editTodo
        deleteTodo
      }}
    />
  );
};
```

A composable design like this keeps the behavior of the model testable. If we had really complex interaction layer behavior, we could unit test it by writing tests against our custom React hook.

```
// hooks/useTodos.specs.tsx
import { useTodos } from "./useTodos";
import { todosVar } from "../cache";

const { operations } = useTodos(todosVar);

describe('useTodos', () => {
  beforeEach(() => {
    // Reset our reactive variables
    todosVar([]);
  });

  it('should add a todo', () => {
    operations.addTodo('First todo')
    expect(
      todosVar()
    ).toHaveLength(1)

    expect(
      todosVar()[0].id
    ).toEqual(0)

    expect(
      todosVar()[0].completed
    ).toEqual(false)

    expect(
      todosVar()[0].text
    ).toEqual('First todo')
  })

  // ...
})
```

Local state management use cases

Here are a few of the most common local state use cases.

Use case #1 — Local only data

This is what we walked through above. Key things to note:

- The reactive variable is used as the source of truth.
- Cache policies call the read function on every field and type before they read from the cache.
- We can query for `local` state using the `@client` directive.
- It's recommended to organize logic that would normally be a mutation away from the presentation layer, preferably in a React hook or another form of organizing model logic.

Use case #2 — Separate local and remote data

This is **the most common use case to encounter.** It's more often that we are working with remote data, and we just need to supplement it with a little bit of additional data that only exists on the client-side.

Let's say you were still building the same `todo` app, but this time, all of the data was sourced from a remote GraphQL API. Our queries would look like this — no `@client` directive.

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

export const GET_ALL_TODOS = gql`
  query GetAllTodos {
    todos {
      id
      text
      completed
    }
  }
`
```

And our client cache policy for the `todos` type on the root `query` object would be removed. We wouldn't need to add any additional config to get this to work.

```
export const cache: InMemoryCache = new InMemoryCache({ });
```

Now let's say you wanted the ability to add *dark mode* to your site (because who doesn't love dark mode)? We'd want to set up a reactive variable specifically for that.

We'd do the same thing, adding a cache policy for the `darkMode` field on the root `Query` type.

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

export const darkModeVar = makeVar<boolean>(false);

export const cache: InMemoryCache = new InMemoryCache({
  typePolicies: {
    Query: {
      fields: {
        darkMode: {
          read () {
            return darkModeVar();
          }
        }
      }
    }
  }
});
```

And if we wanted, we could query for both the current `darkMode` value and the `todos` at the same time from a container component, remembering to specify that `darkMode` is a client-only field.

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

export const GET_TODOS_AND_DARK_MODE = gql`
  query GetTodosAndDarkMode {
    todos {
      id
      text
      completed
    }
    darkMode @client
  }
`
```

Use case #3 — Locally decorated remote data

Sometimes when we're working with remote data, it doesn't contain all the data we need to do our work on the client-side. Sometimes we need to decorate remote data with additional client-side types and fields.

One of the best examples of this is dealing with an `isSelected` field on a list of `todos` retrieved from a GraphQL API. How would we go about hooking that up?

Using cache policies, we can define the `isSelected` field on the `Todo` type. Let's set this to `false` for the time being.

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

export const cache: InMemoryCache = new InMemoryCache({
  typePolicies: {
    Todo: {
      fields: {
        isSelected: {
          read (value, opts) {
            return false;
          }
        }
      }
    }
  }
});
```

To query for this, we can do:

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

export const GET_ALL_TODOS = gql`
  query GetAllTodos {
    todos {
      id
      text
      completed
      isSelected @client
    }
  }
`
```

Now we have to figure out how to determine if a `Todo` is selected or not.

In the `read` function for a cache policy, the second argument gives us utility functions we can use to query into the type under question.

One of the most useful utility functions is `readField`.

Since the `read` function is called for every single `Todo` before it's read from the cache, we can use the `readField` function to ask for the value of a particular field on the type that's being referenced.

That's exactly what we'll do. First, let's get the `id` field of the todo.

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

export const cache: InMemoryCache = new InMemoryCache({
  typePolicies: {
    Todo: {
      fields: {
        isSelected: {
          read (value, { readField }) {
            const todoId = readField('id');

            return false;
          }
        }
      }
    }
  }
});
```

Then let's create a new reactive variable to hold onto the list of currently selected `todo` ids.

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

export const currentSelectedTodoIds = makeVar<number[]>([]);

export const cache: InMemoryCache = new InMemoryCache({
  typePolicies: {
    Todo: {
      fields: {
        isSelected: {
          read (value, { readField }) {
            const todoId = readField('id');

            return isSelected;
          }
        }
      }
    }
  }
});
```

Finally, we can determine if the current `todo` is selected by seeing if it's in the `currentSelectedTodoIds` reactive variable.

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

export const currentSelectedTodoIds = makeVar<number[]>([]);

export const cache: InMemoryCache = new InMemoryCache({
  typePolicies: {
    Todo: {
      fields: {
        isSelected: {
          read (value, { readField }) {
            const todoId = readField('id');
            const isSelected = !!currentSelectedTodoIds()
              .find((id) => id === todoId)

            return isSelected;
          }
        }
      }
    }
  }
});
```

Boom. Now, to *toggle* a todo as selected or not, we can use the same pattern as before, by putting interaction logic for the operation inside of a React hook.

```
// hooks/useTodos.tsx

import { Todo, Todos } from "../../../models/Todos";
import { ReactiveVar } from "@apollo/client";

export function useTodos (
  todosVar: ReactiveVar<Todos>,
  selectedTodoIdsVar: ReactiveVar<number[]>
) {

  ... // Other todos operations

  const toggleTodoSelected = (todoId: number) => {
    const allSelectedTodos = selectedTodoIdsVar();
    const found = !!allSelectedTodos.find((t) => t === todoId);

    if (found) {
      selectedTodosVar(allSelectedTodos.filter((t) => t === todoId))
    } else {
      selectedTodosVar(allSelectedTodos.concat(todoId))
    }
  }

  return {
    operations: { toggleTodoSelected, ... }
  }
}
```

References

https://www.apollographql.com/docs/react/local-state/reactive-variables/

https://www.apollographql.com/blog/apollo-client/architecture/client-side-architecture-basics/
https://www.apollographql.com/blog/apollo-client/caching/local-state-management-with-reactive-variables/
https://khalilstemmler.com/articles/categories/client-side-architecture