# Use Apollo to Manage the App's Local State

#graphql #react #testing #tutorial

**Felipe Armoni** Feb 22, 2020 • *Updated on Jun 23, 2020* • 15 min read

**Managing and Testing Local State with Apollo Graphql and React (3 Part Series)**

1. Creating an App using React and Apollo Graphql
2. **Use Apollo to Manage the App's Local State**
3. Unit Tests with Enzyme and Apollo Graphql

This is a three part tutorial series in which we will build a simple shopping cart app using React and Apollo Graphql. The idea is to build a table in which the user can choose which Rick and Morty action figures he wants to buy.

- Part 1: Creating an App using React and Apollo Graphql.
- Part 2: Use Apollo to manage the app's local state.
- Part 3: Unit Tests with Enzyme and Apollo Graphql

On this second part we will create and manage the local application state using the Apollo In Memory Cache. Our objective is to allow the user to choose how many action figures from the Rick and Morty show he wants to buy and display a checkout screen with the total price and the summary of the chosen items.

This tutorial builds on top of the code generated in the Part 1. You can get it here.

The complete code for the Part 2 is available in this repository and the website here: https://komyg.github.io/rm-shop-v2/.

> Note: this tutorial assumes that you have a working knowledge of React and Typescript.

# Getting Started

To begin, clone the repository that we used on the Part 1.

After you cloned the repository, run `yarn install` to download the necessary packages.

# Creating a local schema

First we will create a local schema to extend the properties that we have on the Rick and Morty API and create new ones. To do this, create a new file called: *local-schema.graphql* inside the *src* folder and paste the code below:

```
type Query {
  shoppingCart: ShoppingCart!
}

type Mutation {
  increaseChosenQuantity(input: ChangeProductQuantity!): Boolean
  decreaseChosenQuantity(input: ChangeProductQuantity!): Boolean
}

extend type Character {
  chosenQuantity: Int!
  unitPrice: Int!
}

type ShoppingCart {
  id: ID!
  totalPrice: Int!
  numActionFigures: Int!
}

input ChangeProductQuantity {
  id: ID!
}
```

Here is the breakdown our local schema:

- As with all Graphql schemas we have the two basic types: `Query` and `Mutation`.
- Inside the `Query` type we added a `shoppingCart` query that will return a `ShoppingCart` object that is stored locally on the Apollo In Memory Cache.

- We also added two mutations: `increaseChosenQuantity` and `decreaseChosenQuantity`. Both will change the quantity the user has chosen for an action figure and update the shopping cart.
- We extended the `Character` type from the Rick and Morty API to add two extra fields: `chosenQuantity` and `unitPrice` that will only exist in our local state.
- We created an `input` type called `ChangeProductQuantity` that will be used inside the mutations. Note that we could send the `characterId` directly to the mutation, but we created the `input` type to illustrate its use. Also, a query or mutation can only accept a `scalar` or an `input` types as its arguments. They do not support regular `types`.

> Note: the exclamation mark (`!`) at the end of the types, indicates that they are obligatory, therefore they cannot be null or undefined.

## Updating the Grapqhql Codegen config file

Update the *codegen.yml* file to include the local schema we just created. We are also going to add the fragment matcher generator, so that we can use fragments on our queries and mutations.

```
overwrite: true
schema: "https://rickandmortyapi.com/graphql"
documents: "src/**/*.graphql"
generates:
  src/generated/graphql.tsx:
    schema: "./src/local-schema.graphql" # Local Schema
    plugins:
      - "typescript"
      - "typescript-operations"
      - "typescript-react-apollo"
      - "fragment-matcher"
```

```
    # Add this to use hooks:
    config:
      withHooks: true

  # Fragment Matcher
  src/generated/fragment-matcher.json:
    schema: "./src/local-schema.graphql"
    plugins:
      - "fragment-matcher"
```

## Creating an initial state

When our application loads, it is good to initialize Apollo's `InMemoryCache` with an initial state based on our local schema. To do this, let's add the `initLocalCache` function to the *config/apollo-local-cache.ts* file:

```
export function initLocalCache() {
  localCache.writeData({
    data: {
      shoppingCart: {
        __typename: 'ShoppingCart',
        id: btoa('ShoppingCart:1'),
        totalPrice: 0,
        numActionFigures: 0,
      },
    },
  });
}
```

Here we are initializing the `ShoppingCart` objet with default values. Also note that we using an ID pattern of `[Typename]:[ID]` encoded in base 64. You can use this or any other pattern you like for the ID's as long as they are always unique.

Also note that it if we chose not to initialize the `ShoppingCart` object, it would be better to set it as `null` instead of leaving it as `undefined`. This is to avoid errors when running the `readQuery` function on the Apollo's `InMemoryCache`. If the object we are querying is `undefined`, then the `readQuery` will throw an error, but if it is `null`, then it will return `null` without throwing an exception.

Initializing the `ShoppingCart` to `null` would look like this:

```
// Don't forget that in this tutorial we want to have the shoppingCart initialized, so don't c
export function initLocalCache() {
  localCache.writeData({
    data: {
      shoppingCart: null,
  });
}
```

Now lets call the `initLocalCache` function after the Apollo Client has been initialized in the *config/apollo-client.ts* file:

```
export const apolloClient = new ApolloClient({
  link: ApolloLink.from([errorLink, httpLink]),
  connectToDevTools: process.env.NODE_ENV !== 'production',
  cache: localCache,
```

```
  assumeImmutableResults: true,
});

initLocalCache();
```

# Creating resolvers

Resolvers are functions that will manage our local `InMemoryCache`, by reading data from it and writing data to it. If you are accustomed to Redux, the resolvers would be similar to the reducer functions, even though they are not required to be synchronous nor are the changes to the `InMemoryCache` required to be immutable, although we chose to use immutability in the Part 1 of this tutorial in return for performance improvements.

## Type resolvers

Type resolvers are used to initialize the local fields of a remote type. In our case, we have extended the `Character` type with the `chosenQuantity` and `unitPrice` fields.

To start, create the *src/resolvers* folder. Then create the *set-unit-price.resolver.ts* file and copy the contents below:

```
import ApolloClient from 'apollo-client';
import { Character } from '../generated/graphql';
import { InMemoryCache } from 'apollo-cache-inmemory';

export default function setChosenQuantity(
  root: Character,
```

```
  variables: any,
  context: { cache: InMemoryCache; getCacheKey: any; client: ApolloClient<any> },
  info: any
) {
  switch (root.name) {
    case 'Rick Sanchez':
      return 10;

    case 'Morty Smith':
      return 10;

    default:
      return 5;
  }
}
```

This resolver will receive each character from the backend and assign it unit price based on the character's name.

Then, lets connect this resolver our client. To do this, create the file: *config/apollo-resolvers.ts* and paste the contents below:

```
import setUnitPrice from '../resolvers/set-unit-price.resolver';

export const localResolvers = {
  Character: {
    chosenQuantity: () => 0,
    unitPrice: setUnitPrice,
  },
```

```
};
```

Since the initial value for the `chosenQuantity` will always be 0, then we will just create a function that returns 0.

Then, add the `localResolvers` to our client config in: *config/apollo-client.ts*.

```
export const apolloClient = new ApolloClient({
  link: ApolloLink.from([errorLink, httpLink]),
  connectToDevTools: process.env.NODE_ENV !== 'production',
  cache: localCache,
  assumeImmutableResults: true,
  resolvers: localResolvers,
});

initLocalCache();
```

## Creating local queries

Now we can create a new query that will return the `ShoppingCart` object. To do this, create a new file called: *graphql/get-shopping-cart.query.graphql* and paste the contents below:

```
query GetShoppingCart {
  shoppingCart @client {
    id
    __typename
```

```
    totalPrice
    numActionFigures
  }
}
```

Now run the `yarn gen-graphql` command to generate its types. Notice that we can get the `ShoppingCart` without having to create a resolver, because the `ShoppingCart` object is a direct child of the root query.

## Mutation resolvers

Now we are going to create mutations that will handle increasing and decreasing the quantity of a `Character`. First we should create a graphql file that will describe the mutation. Create the file: *graphql/increase-chosen-quantity.mutation.graphql* and paste the contents below:

```
mutation IncreaseChosenQuantity($input: ChangeProductQuantity!) {
  increaseChosenQuantity(input: $input) @client
}
```

Here we are using the `@client` annotation to indicate that this mutation should be ran locally on the `InMemoryCache`.

Also create another file: *graphql/decrease-chosen-quantity.mutation.graphql* and paste the contents below:

```
mutation DecreaseChosenQuantity($input: ChangeProductQuantity!) {
  decreaseChosenQuantity(input: $input) @client
}
```

Finally, let's also create a fragment that will be useful to retrieve a single `Character` directly from the cache. In Graphql a fragment is a pice of code that can be reused in queries and mutations. It can also be used to retrieve and update data directly in the Apollo's `InMemoryCache` without having to go through the root query.

This means that through the fragment below, we can get a single `Character` using its `__typename` and `id`.

> Note: here we should have used the `character(id: ID)` query that is available in the graphql server, but I preferred to do this locally to demonstrate how it is done.

Create the *graphql/character-data.fragment.graphql* file:

```
fragment characterData on Character {
  id
  __typename
  name
  unitPrice @client
  chosenQuantity @client
}
```

Now run the Graphql Code Gen command to update our generated files: `yarn gen-graphql`. Then update the *config/apollo-local-cache.ts* with the fragment matcher:

```
import { InMemoryCache, IntrospectionFragmentMatcher } from 'apollo-cache-inmemory';
import introspectionQueryResultData from '../generated/fragment-matcher.json';
```

```
export const localCache = new InMemoryCache({
  fragmentMatcher: new IntrospectionFragmentMatcher({ introspectionQueryResultData }),
  freezeResults: true,
});

export function initLocalCache() {
  localCache.writeData({
    data: {
      shoppingCart: {
        __typename: 'ShoppingCart',
        id: btoa('ShoppingCart:1'),
        totalPrice: 0,
        numActionFigures: 0,
      },
    },
  });
}
```

Now let's create the resolvers themselves. First create the *resolvers/increase-chosen-quantity.resolver.ts*:

```
import ApolloClient from 'apollo-client';
import { InMemoryCache } from 'apollo-cache-inmemory';
import {
  CharacterDataFragment,
  CharacterDataFragmentDoc,
  IncreaseChosenQuantityMutationVariables,
  GetShoppingCartQuery,
  GetShoppingCartDocument,
} from '../generated/graphql';
```

```
export default function increaseChosenQuantity(
  root: any,
  variables: IncreaseChosenQuantityMutationVariables,
  context: { cache: InMemoryCache; getCacheKey: any; client: ApolloClient<any> },
  info: any
) {
  const character = getCharacterFromCache(variables.input.id, context.cache, context.getCacheK
  if (!character) {
    return false;
  }

  updateCharacter(character, context.cache, context.getCacheKey);
  updateShoppingCart(character, context.cache);

  return true;
}

function getCharacterFromCache(id: string, cache: InMemoryCache, getCacheKey: any) {
  return cache.readFragment<CharacterDataFragment>({
    fragment: CharacterDataFragmentDoc,
    id: getCacheKey({ id, __typename: 'Character' }),
  });
}

function updateCharacter(character: CharacterDataFragment, cache: InMemoryCache, getCacheKey:
  cache.writeFragment<CharacterDataFragment>({
    fragment: CharacterDataFragmentDoc,
    id: getCacheKey({ id: character.id, __typename: 'Character' }),
    data: {
      ...character,
      chosenQuantity: character.chosenQuantity + 1,
    },
```

```
  });
}

function updateShoppingCart(character: CharacterDataFragment, cache: InMemoryCache) {
  const shoppingCart = getShoppingCart(cache);
  if (!shoppingCart) {
    return false;
  }

  cache.writeQuery<GetShoppingCartQuery>({
    query: GetShoppingCartDocument,
    data: {
      shoppingCart: {
        ...shoppingCart,
        numActionFigures: shoppingCart.numActionFigures + 1,
        totalPrice: shoppingCart.totalPrice + character.unitPrice,
      },
    },
  });
}

function getShoppingCart(cache: InMemoryCache) {
  const query = cache.readQuery<GetShoppingCartQuery>({
    query: GetShoppingCartDocument,
  });

  return query?.shoppingCart;
}
```

There is quite a bit happening here:

- First we have the `getCharacterFromCache` function that retrieves a `Character` from the cache using the `CharacterData` fragment. This way we can retrieve the character directly, instead of having to go through the root query.
- Then we have the `updateCharacter` function that increases the chosen quantity for this character by one. Notice that we are using the same `CharacterData` fragment to update the cache and that we are not updating the character directly, instead we are using the spread operator to update the cache with a copy of the original `Character` object. We've done this, because we decided to use immutable objects.
- Then we update the `ShoppingCart`, by using the `GetShoppingCartQuery` to get the current state of the `ShoppingCart` and update the number of chosen `Characters` and the total price. Here we can use a query to retrieve the `ShoppingCart`, because it is a child of the root query, so we can get it directly.
- When using fragments, we use the `getCacheKey` function to get an object's cache key. By default, the Apollo Client stores the data in a de-normalized fashion, so that we can use fragments and the cache key to access any object directly. Usually each cache key is composed as `__typename:id`, but it is a good practice to use the `getCacheKey` function in case you want to use a custom function to create the cache keys.
- Notice that we are using the `readQuery` function to retrieve the current state of the `ShoppingCart`. We can do this, because we have set the initial state for the shopping cart, however if we had not set it, then this function would throw an exception the first time it ran, because its result would be `undefined`. If you do not want to set a definite state for a cache object, then it is good to set its initial state as `null`, instead of leaving it as `undefined`. This way, when you execute the `readQuery` function it will not throw an exception.
- It is also worth mentioning, that we could use the `client.query` function instead of the `cache.readQuery`, this way we would not have to worry about the `ShoppingCart` being `undefined`, because the `client.query` function does not throw an error if the object it wants to retrieve is `undefined`. However the `cache.readQuery` is faster and it is also synchronous (which is useful in this context).

- It is also worth mentioning that whenever we write data to the `InMemoryCache` using either the `writeQuery` or the `writeFragment` functions, than only the fields that are specified in the query or the fragment are updated, all other fields are ignored. So we wouldn't be able to update a character's image by using the `characterData` fragment, because the image parameter is not specified on it.

Now we will create a new resolver to decrease a `Character` chosen quantity. Please create the file: *resolvers/decrease-chosen-quantity.resolver.ts* and copy and paste the contents below:

```
import ApolloClient from 'apollo-client';
import { InMemoryCache } from 'apollo-cache-inmemory';
import {
  CharacterDataFragment,
  CharacterDataFragmentDoc,
  IncreaseChosenQuantityMutationVariables,
  GetShoppingCartQuery,
  GetShoppingCartDocument,
} from '../generated/graphql';

export default function decreaseChosenQuantity(
  root: any,
  variables: IncreaseChosenQuantityMutationVariables,
  context: { cache: InMemoryCache; getCacheKey: any; client: ApolloClient<any> },
  info: any
) {
  const character = getCharacterFromCache(variables.input.id, context.cache, context.getCacheK
  if (!character) {
    return false;
  }
```

```
  updateCharacter(character, context.cache, context.getCacheKey);
  updateShoppingCart(character, context.cache);

  return true;
}

function getCharacterFromCache(id: string, cache: InMemoryCache, getCacheKey: any) {
  return cache.readFragment<CharacterDataFragment>({
    fragment: CharacterDataFragmentDoc,
    id: getCacheKey({ id, __typename: 'Character' }),
  });
}

function updateCharacter(character: CharacterDataFragment, cache: InMemoryCache, getCacheKey:
  let quantity = character.chosenQuantity - 1;
  if (quantity < 0) {
    quantity = 0;
  }

  cache.writeFragment<CharacterDataFragment>({
    fragment: CharacterDataFragmentDoc,
    id: getCacheKey({ id: character.id, __typename: 'Character' }),
    data: {
      ...character,
      chosenQuantity: quantity,
    },
  });
}

function updateShoppingCart(character: CharacterDataFragment, cache: InMemoryCache) {
  const shoppingCart = getShoppingCart(cache);
  if (!shoppingCart) {
```

```
    return false;
  }

  let quantity = shoppingCart.numActionFigures - 1;
  if (quantity < 0) {
    quantity = 0;
  }

  let price = shoppingCart.totalPrice - character.unitPrice;
  if (price < 0) {
    price = 0;
  }

  cache.writeQuery<GetShoppingCartQuery>({
    query: GetShoppingCartDocument,
    data: {
      shoppingCart: {
        ...shoppingCart,
        numActionFigures: quantity,
        totalPrice: price,
      },
    },
  });
}

function getShoppingCart(cache: InMemoryCache) {
  const query = cache.readQuery<GetShoppingCartQuery>({
    query: GetShoppingCartDocument,
  });

  return query?.shoppingCart;
}
```

This resolver is very similar to the other one, with the exception that we do not allow the quantities and the total price to be less than 0.

Finally let's connect these two resolvers to the Apollo client, by updating the *config/apollo-resolvers.ts* file:

```
import setUnitPrice from '../resolvers/set-unit-price.resolver';
import increaseChosenQuantity from '../resolvers/increase-chosen-quantity.resolver';
import decreaseChosenQuantity from '../resolvers/decrease-chosen-quantity.resolver';

export const localResolvers = {
  Mutations: {
    increaseChosenQuantity,
    decreaseChosenQuantity,
  },
  Character: {
    chosenQuantity: () => 0,
    unitPrice: setUnitPrice,
  },
};
```

## Query resolvers

Technically we won't be needing any query resolvers for this app, but I think that it might be useful to do an example. So we are going to create a resolver that will return the data available for a `Character`.

> Note that in a real project we should use the `character(id: ID)` query that is already available from the server instead of creating a new query.

To begin, update the `Query` type in our local schema:

```
type Query {
  shoppingCart: ShoppingCart!
  getCharacter(id: ID!): Character
}
```

Now, create a new file called: *graphql/get-character.query.graphql* and paste the contents below:

```
query GetCharacter($id: ID!) {
  getCharacter(id: $id) @client {
    ...characterData
  }
}
```

Now re-generate the graphql files with the command: `yarn gen-graphql`.

For the resolver itself, create a new file called: *resolvers/get-character.resolver.ts*:

```
import { InMemoryCache } from 'apollo-cache-inmemory';
import ApolloClient from 'apollo-client';
import {
```

```
  CharacterDataFragmentDoc,
  CharacterDataFragment,
  GetCharacterQueryVariables,
} from '../generated/graphql';

export default function getCharacter(
  root: any,
  variables: GetCharacterQueryVariables,
  context: { cache: InMemoryCache; getCacheKey: any; client: ApolloClient<any> },
  info: any
) {
  return context.cache.readFragment<CharacterDataFragment>({
    fragment: CharacterDataFragmentDoc,
    id: context.getCacheKey({ id: variables.id, __typename: 'Character' }),
  });
}
```

Finally let's connect this new resolver to the Apollo client by updating the *config/apollo-resolvers.ts* file:

```
import setUnitPrice from '../resolvers/set-unit-price.resolver';
import increaseChosenQuantity from '../resolvers/increase-chosen-quantity.resolver';
import decreaseChosenQuantity from '../resolvers/decrease-chosen-quantity.resolver';
import getCharacter from '../resolvers/get-character.resolver';

export const localResolvers = {
  Query: {
    getCharacter,
  },
  Mutation: {
```

```
    increaseChosenQuantity,
    decreaseChosenQuantity,
  },
  Character: {
    chosenQuantity: () => 0,
    unitPrice: setUnitPrice,
  },
};
```

## Updating our components

Now that we have created our mutations and resolvers we will update our components to use them. First let's update our `GetCharactersQuery` to include our new local fields. Open the *graphql/get-characters.query.graphql* file and paste the contents below:

```
query GetCharacters {
  characters {
    __typename
    results {
      id
      __typename
      name
      image
      species
      chosenQuantity @client
      unitPrice @client
      origin {
        id
        __typename
```

```
          name
        }
        location {
          id
          __typename
          name
        }
      }
    }
  }
```

Here we added the `chosenQuantity` and `unitPrice` fields with the `@client` annotation to tell Apollo that these fields are used only on the client.

Don't forget to regenerate our graphql types by running the `yarn gen-graphql` command on your console.

Now let's update our table to add these new fields. First open the *components/character-table/character-table.tsx* file and add two more columns to our table, one for the unit price and the other for the chosen quantity:

```
// Display the data
return (
  <TableContainer component={Paper}>
    <Table>
      <TableHead>
        <TableRow>
          <TableCell>
            <strong>Name</strong>
```

```
            </TableCell>
            <TableCell>
              <strong>Species</strong>
            </TableCell>
            <TableCell>
              <strong>Origin</strong>
            </TableCell>
            <TableCell>
              <strong>Location</strong>
            </TableCell>
            <TableCell>
              <strong>Price</strong>
            </TableCell>
            <TableCell>
              <strong>Quantity</strong>
            </TableCell>
          </TableRow>
        </TableHead>
        <TableBody>
          {data.characters.results.map(character => (
            <CharacterData character={character} key={character?.id!} />
          ))}
        </TableBody>
      </Table>
    </TableContainer>
  );
  );
```

Now we are going to create a new component to handle the user's choices. First add the Material UI Icons package: `yarn add @material-ui/icons`. Then create the file: *components/character-quantity/character-*

*quantity.tsx* and paste the contents below:

```
import React, { ReactElement, useCallback } from 'react';
import { Box, IconButton, Typography } from '@material-ui/core';
import ChevronLeftIcon from '@material-ui/icons/ChevronLeft';
import ChevronRightIcon from '@material-ui/icons/ChevronRight';
import {
  useIncreaseChosenQuantityMutation,
  useDecreaseChosenQuantityMutation,
} from '../../generated/graphql';

interface Props {
  characterId: string;
  chosenQuantity: number;
}

export default function CharacterQuantity(props: Props): ReactElement {
  // Mutation Hooks
  const [increaseQty] = useIncreaseChosenQuantityMutation({
    variables: { input: { id: props.characterId } },
  });
  const [decreaseQty] = useDecreaseChosenQuantityMutation();

  // Callbacks
  const onIncreaseQty = useCallback(() => {
    increaseQty();
  }, [increaseQty]);
  const onDecreaseQty = useCallback(() => {
    decreaseQty({ variables: { input: { id: props.characterId } } });
  }, [props.characterId, decreaseQty]);
```

```
  return (
    <Box display='flex' alignItems='center'>
      <IconButton color='primary' disabled={props.chosenQuantity <= 0} onClick={onDecreaseQty}>
        <ChevronLeftIcon />
      </IconButton>
      <Typography>{props.chosenQuantity}</Typography>
      <IconButton color='primary' onClick={onIncreaseQty}>
        <ChevronRightIcon />
      </IconButton>
    </Box>
  );
}
```

In this component we are using two hooks to instantiate our mutations and then we are using two callbacks to call them whenever the user clicks on the increase or decrease quantity buttons.

You will notice that we've set the input for the `useIncreaseChosenQuantityMutation` when it was first instantiated and that we've set the input for the `useDecreaseChosenQuantityMutation` on the callback. Both options will work in this context, but it is worth saying that the input defined on the first mutation is static, and the input defined on the second mutation is dynamic. So, if we were working with a form for example, then we should have chosen to set the mutation's input when it is called not when it is first instantiated, otherwise it will always be called with our form's initial values.

Also there is no need to call another query here to get the character's chosen quantity, because this value already comes from the query we made in the `CharacterTable` component and it will be automatically updated by Apollo and passed down to this component when we fire the mutations.

Now open the file: *components/character-data/character-data.tsx* and include our new fields:

```
export default function CharacterData(props: Props): ReactElement {
  const classes = useStyles();

  return (
    <TableRow>
      <TableCell className={classes.nameTableCell}>
        <Box>
          <img src={props.character?.image!} alt='' className={classes.characterImg} />
        </Box>
        <Typography variant='body2' className={classes.characterName}>
          {props.character?.name}
        </Typography>
      </TableCell>
      <TableCell>{props.character?.species}</TableCell>
      <TableCell>{props.character?.origin?.name}</TableCell>
      <TableCell>{props.character?.location?.name}</TableCell>
      <TableCell>{props.character?.unitPrice}</TableCell>
      <TableCell>
        <CharacterQuantity
          characterId={props.character?.id!}
          chosenQuantity={props.character?.chosenQuantity!}
        />
      </TableCell>
    </TableRow>
  );
}
```

Now run our project using the `yarn start` command. You should see the unit price we set for each character (Rick and Morty should have a higher price than the others) and you should be able to increase and decrease each character's chosen quantity.

## The Shopping Cart

Now let's add a shopping cart component that will show the total price and the total number of action figures that were chosen by the user. To do this, create a new component: *components/shopping-cart-btn/shopping-cart-btn.tsx* and paste the content below:

```
import React, { ReactElement } from 'react';
import { Fab, Box, makeStyles, createStyles, Theme, Typography } from '@material-ui/core';
import { useGetShoppingCartQuery } from '../../generated/graphql';
import ShoppingCartIcon from '@material-ui/icons/ShoppingCart';

const useStyles = makeStyles((theme: Theme) =>
  createStyles({
    root: {
      position: 'fixed',
      bottom: theme.spacing(4),
    },
    quantityText: {
      position: 'absolute',
      top: '4px',
      left: '50px',
      color: 'white',
    },
    btnElement: {
      padding: theme.spacing(1),
```

```
    },
  })
);

export default function ShoppingCartBtn(): ReactElement {
  const classes = useStyles();
  const { data } = useGetShoppingCartQuery();

  if (!data || data.shoppingCart.numActionFigures <= 0) {
    return <Box className={classes.root} />;
  }

  return (
    <Box className={classes.root}>
      <Fab variant='extended' color='primary'>
        <Box>
          <ShoppingCartIcon className={classes.btnElement} />
          <Typography variant='caption' className={classes.quantityText}>
            {data.shoppingCart.numActionFigures}
          </Typography>
        </Box>

        <Typography className={classes.btnElement}>
          {formatPrice(data.shoppingCart.totalPrice)}
        </Typography>
      </Fab>
    </Box>
  );
}

function formatPrice(price: number) {
```

```
  return `US$ ${price.toFixed(2)}`;
}
```

In this component we are using the `useGetShoppingCart` query hook to get the number of action figures that the user selected and the total price. The state of the `ShoppingCart` is handled on the Apollo `InMemoryCache` and is updated whenever we increase or decrease the action figure's quantities by their respective resolvers. We are also hiding this component until the customer has chosen at least one action figure.

Notice that we didn't needed to create a resolver to get the shopping cart's state. That is because the shopping cart's state is available as a direct child of the root Query, therefore we can get it more easily.

> Note: in a real project, this button would take the user to some kind of checkout screen in which he would be able to review and place his order.

Finally let's update our app component to contain our new button. To do this, open the *components/app/app.tsx* file and add the `ShoppingCartBtn` component:

```
export default function App(): ReactElement {
  const classes = useStyles();

  return (
    <Container className={classes.root}>
      <Box display='flex' justifyContent='center' alignContent='center'>
        <CharacterTable />
        <ShoppingCartBtn />
      </Box>
    </Container>
```

```
    );
  }
```

## Conclusion

If all goes well, when you run our app you should be able to increase and decrease the desired quantity of action figures and see the total number and total price of the chosen products.

**Managing and Testing Local State with Apollo Graphql and React (3 Part Series)**

1. Creating an App using React and Apollo Graphql
2. **Use Apollo to Manage the App's Local State**
3. Unit Tests with Enzyme and Apollo Graphql

## Discussion (6)

Subscribe

DEV

Add to the discussion

**1awaleed** • Dec 14 '20 • Edited on Dec 17

Thanks a lot @felipe for the awesome article

I followed what you have done to the teeth but I used typePolicies instead of resolvers as peter noted

I need your help in a small question, If I need to overwrite a type in the remote scheme? what should I do?

For example, my remote schema has:
type foo{
bar: string
}

but I want to change that to

type foo{
bar: boolean
}

I am manipulating the result of this field in the `read` function of the typePolicy.

Thanks again for the awesome read. Been super helpful.

1 like Reply

**Felipe Armoni** • Dec 14 '20

Hi,

I am glad that the article was helpful! I do have to update it to Apollo v3, but I haven't had the time yet.

As for your question, I am not sure if you can overwrite a field that comes from your remote schema. Did you try to add the `@client` annotation on the field? Did that trigger your read function? If not, maybe you can try creating a resolver for this particular field.

We had a problem a little while ago, in which we had an object that did not have an ID field. We wanted to add it in the client so that we could use the `readFragment` and `writeFragment` functions from Apollo and optimize our data handling.

We tried to use a read function for this, but it wasn't working, so we had to create a custom resolver for it, even though we are using Apollo 3. I am not sure if this was because the ID field is a special one, since it is used in the cache denormalisation, but it was the recommendation that we received from the Apollo dev team.

Thanks,
Felipe

2 likes Reply

• Dec 17 '20 • Edited on Dec 17

**1awaleed**

Thanks @felipe for your reply.

Adding @client should when I am using a value as a client local state but this is not what I am trying todo. Using @client works great. I am using the typePolicy read function to read a string field `bar` and returning either true or false. The functionality is working fine but typescript is complaining 😭 The remote type used here with introspection is string but I want to use it everywhere in the code as boolean. This is called scheme stitching (I think?)

Regarding your problem, I think the problem is with the ID but no worries, In apollo/client v3, Now you can set a function that defines & returns unique keys for specific types without the need for resolvers. You can read more about it here: apollographql.com/docs/react/cachi...

I have a small question regarding how you use `readFragment` and `writeFragment`.
does `readFragment` reads sibling fields of the same type? Another thing is what would happen if you read a field of an instance of an object that was never fetched before? does it crash or throw error or try to fetch that field?

So far, Apollo's experience have been great and I am already seeing a lot of benefits using it.

Thanks
A. Waleed

Like *Thread*

**Felipe Armoni** • Dec 20 '20

Hi @1awaleed ,

Thanks for the tip regarding the ID. I will look into that.

As for your schema stiching problem, if you are using the Graphql Codegen, you could try to extend the remote object using something like this:

```
extend type Product {
  quantity: Int!
  isValid: Boolean!
}
```

But I have only used this to add extra fields to an existing type, not to override existing ones. On that subject, is it absolutely necessary to override the remote field? Can't you create a local boolean field that uses the remote one as a reference?

As for the `readFragment` and `writeFragment` question: yes, you use them with sibling fields and with any other type that has an `ID` and a `__typename`. For example:

```
{
  "productEdge": {
    "id": "UHJvZHVjdEVkZ2U6MQ==",
    "__typename": "ProductEdge",
    "edges": [
```

```
      {
        "node": {
          "id": "UHJvZHVjdDox",
          "__typename": "Product",
          "numericalId": 1,
          "name": "Product 1"
        }
      },
      {
        "node": {
          "id": "UHJvZHVjdDoy",
          "__typename": "Product",
          "numericalId": 2,
          "name": "Product 2"
        }
      }
    ]
  }
}
```

You can read and write the products above like this:

```
const productFragment = gql`
  fragment product on Product {
    numericalId
    name
  }
`

const product1 = client.readFragment({ fragment: productFragment, id: 'Product:
```

```
const product2 = client.readFragment({ fragment: productFragment, id: 'Product:

client.writeFragment({ fragment: productFragment, id: 'Product:UHJvZHVjdDox', d
```

So you can read and write to sibling fields using `readFragment` and `writeFragment`, but since you have to pass the `ID` of the object you want, you can only read and write to one at the time.

If you want to change two or more siblings at the same time, I would recommend using either using a query or a fragment that retrieves a higher level element, for example:

```
const productQuery = gql`
  query ProductQuery {
    productsEdge {
      edges {
        node {
          numericalId
          name
        }
      }
    }
  }
`;

const productsFragment = gql`
  fragment products on ProductEdge {
    edges {
      node {
```

```
          numericalId
          name
        }
      }
    }
  `;
```

I hope this answers you question.

Thanks,
Felipe

2 likes  *Thread*

**1awaleed** • Dec 24 '20

Thanks a lot @ Felipe, you have been helped a lot.

2 likes  Reply

**Peter Kellner** • Oct 25 '20

You might want to put a note on top saying that Apollo is obsoleting local resolvers and removing them from the stack in the future.

1 like  Reply

## Read next

**React Practice Project for Beginner to Advance**

Akhil - Sep 1

**Simple NFT images Generator with Javascript Nodejs (step by step)**

Victor Quan Lam - Sep 5

**How to post data from React to Flask.**

Ondiek Elijah - Sep 4

**Learn Javascript by making an Image Slideshow**

Ramiro - Sep 6

**Felipe Armoni**

Follow

LOCATION
São Paulo

WORK
Senior Engineer at Runa HR

JOINED
Jan 5, 2020

## More from Felipe Armoni

Angular Fire and Forget Polling with NgRx, RxJS and Unit Tests

#angular #ngrx #tutorial #testing

Unit Tests with Enzyme and Apollo Graphql

#graphl #react #testing #tutorial

Creating an App using React and Apollo Graphql

#graphql #react #testing #tutorial