Creating a React Native app using Apollo Graphql v3

#graphql #reactnative #testing #tutorial

Harrison Henri dos Santos Nascimento Jun 27 • 12 min read

Creating an React Native app using Apollo Graphql v3 (2 Part Series)

1. **Creating a React Native app using Apollo Graphql v3**
2. Unit tests with react-native testing library and Apollo Grap...

Introduction

In this tutorial we will build a react-native shopping cart app using the version 3 of Apollo Graphql. This tutorial is based on these great three part series of articles focusing on Apollo v3 and a basic structure of projects using this technology stack.

Note: this tutorial assumes that you have a working knowledge of React-native, typescript and node.

The final source code of this tutorial can be obtained by acessing https://github.com/HarrisonHenri/rick-morty-react-native-shop.

Beginning

Firstly, we create a brand new react-native app using react-native cli:

```
npx react-native init rickmortyshop --template react-native-template-typescript
```

and then, we remove the *App.tsx* file, add a new *index.tsx* file at *src* and modify the content of *index.js* to:

```
/**
 * @format
 */

import { AppRegistry } from 'react-native';
import App from './src/index';
import { name as appName } from './app.json';

AppRegistry.registerComponent(appName, () => App);
```

Apollo graphql installation and setup

Now we will Add the necessary packages from apollo:

```
yarn add graphql @apollo/client graphql-tag
```

The setup is very simillar to presented at https://dev.to/komyg/creating-an-app-using-react-and-apollo-graphql-1ine, so I'll describe this part very succinctly:

Apollo client

To create our connection between the client app and the Graphql server at *src/common/config/apollo/http-link.ts* we write:

```
import { HttpLink } from '@apollo/client/link/http';

export function createHttpLink() {
  return new HttpLink({
    uri: 'https://rickandmortyapi.com/graphql',
  });
}
```

Error link

Now, to to handle errors at our requests at *src/common/config/apollo/error-link.ts* we write:

```
import { onError } from '@apollo/client/link/error';

export const errorLink = onError(({ graphQLErrors, networkError, response, operation }) => {
  if (graphQLErrors) {
    for (const error of graphQLErrors) {
```

```
        console.error(
          `[GraphQL error]: Message: ${error.message}, Location: ${error.locations}, Path: ${err
          operation,
          response
        );
      }
    }
    if (networkError) {
      console.error(`[Network error]: ${networkError}`, operation, response);
    }
  });
```

Apollo local cache

Here, different from https://dev.to/komyg/creating-an-app-using-react-and-apollo-graphql-1ine, we just write (at *src/common/config/apollo/local-cache.ts*):

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

export const localCache = new InMemoryCache();
```

That's why, we don't need to `freezeResults` anymore because now this is the default behavior.

Wrappping all

At *src/common/apollo-client.ts* we write:

```
import { ApolloClient, ApolloLink } from '@apollo/client';
import { errorLink } from './apollo/error-link';
import { createHttpLink } from './apollo/http-link';
import { localCache } from './apollo/local-cache';

export function createApolloClient() {
  const httpLink = createHttpLink();

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

  return apolloClient;
}
```

Here we are putting everything together and creating our `ApolloClient` object. Now at `src/index.tsx` we write:

```
import React from 'react';
import { ApolloProvider } from '@apollo/client';
import { Text, View } from 'react-native';
import { createApolloClient } from './common/config/apollo-client';

const apolloClient = createApolloClient();
```

```
const App = () => {
  return (
    <ApolloProvider client={apolloClient}>
      <View>
        <Text>Hello</Text>
      </View>
    </ApolloProvider>
  );
};

export default App;
```

in order to make the ApolloProvider available globally.

Graphql codegen

Now we will install and setup the graphql codegen,a tool that automatically generates typescript types based on the server schema. After installing, execute the cli using:

```
yarn graphql-codegen init
```

then, follow the steps bellow:

1. Application built with React.
2. Type the Rick and Morty Graphql api url: https://rickandmortyapi.com/graphql.
3. Use the default value (`src/**/*.graphql`) for the fragment and operations.

4. Then pick the following plugins: `TypeScript`, `TypeScript Operators`, `TypeScript React Apollo` and `Introspection Fragment Matcher`.
5. Type the following value: `src/common/generated/graphql.tsx`.
6. Answer no when it asks if you want to generate an introspection file.
7. Use the default value for the name of the config file.
8. Type in `gen-graphql` when it asks the name of the script in the package.json that will be used to generate the graphql files.

and the `yarn install` to install all necessary plugins.

Creating the home screen

First, we will create our first query to retrieve all characters from Rick and Morty cartoon. To achieve this, at *src/common/graphql/queries/get-characters.query.graphql* we paste:

```
query GetCharacters {
  characters {
    __typename
    results {
      id
      __typename
      name
      image
      species
      origin {
        id
        __typename
        name
```

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

Then we run:

```
yarn gen-graphql
```

and if the command runs successfully, you should see that a *graphql.tsx* file was created. Now, at *src/common/components/CharacterCard.tsx* we paste the code bellow:

```
import React from 'react';
import { Image, StyleSheet, Text, View } from 'react-native';

interface Props {
  data: {
    image?: string | null;
    name?: string | null;
  };
}
```

```
const CharacterCard: React.FC<Props> = ({ data }) => {
   return (
    <View style={styles.container}>
      {data.image && (
        <Image source={{ uri: data.image }} style={styles.image} />
      )}
      <View style={styles.details}>
        <Text style={styles.text}>{data.name}</Text>
      </View>
    </View>
  );
};

export default CharacterCard;

const styles = StyleSheet.create({
  container: {
    width: '100%',
    borderRadius: 20,
    marginVertical: 8,
    paddingHorizontal: 8,
    paddingVertical: 24,
    backgroundColor: '#F0F0F0',
    flexDirection: 'row',
  },
  image: { width: 70, height: 70 },
  details: {
    marginLeft: 8,
    justifyContent: 'space-between',
    flex: 1,
  },
  text: {
```

```
    fontSize: 16,
    fontWeight: 'bold',
  },
});
```

Now, at *scr/screens/Home.tsx* we write:

```
import React from 'react';
import { ActivityIndicator, FlatList, StyleSheet, View } from 'react-native';
import { Character, useGetCharactersQuery } from '../common/generated/graphql';

import CharacterCard from '../common/components/CharacterCard';

const Home = () => {
  const { data, loading } = useGetCharactersQuery();

  if (loading) {
    return (
      <View style={styles.container}>
        <ActivityIndicator color="#32B768" size="large" />
      </View>
    );
  }

  return (
    <View style={styles.container}>
      <FlatList
        data={data?.characters?.results}
        renderItem={({ item }) => <CharacterCard data={item as Character} />}
        contentContainerStyle={styles.characterList}
```

```
      />
    </View>
  );
};

export default Home;

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#FFFFFF',
  },
  characterList: {
    padding: 16,
  },
});
```

Here we are using the `useGetCharactersQuery` hook provided by the graphql codegen. The hook provided useful fetch tools like the server response (`data`) and the `loading` state. Then, finally at *src/index.tsx* we write:

```
import React from 'react';
import { ApolloProvider } from '@apollo/client';
import { createApolloClient } from './common/config/apollo-client';

import Home from './screens/Home';

const apolloClient = createApolloClient();
```

```
const App = () => {
  return (
    <ApolloProvider client={apolloClient}>
      <Home />
    </ApolloProvider>
  );
};

export default App;
```

If everything goes well, when you run `yarn start` you will se your app cards with all Ricky Morty characters presented!

Creating the shopping cart logic

Updating queries

Now that we are retrieving the data from the remote server we can start to create the shopping cart logic. First we will write our local-shema at *src/common/config/local-schema.graphql*:

```
type Query {
  shoppingCart: ShoppingCart!
}

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

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

And then, at *src/common/graphql/fragments/character.fragment.graphql* we write:

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

Here we create this fragment to insert @client fields (fields that only exists at our local cache state) and also to use this fragment at our character query. So, in order to do this, we update the *src/common/graphql/queries/get-characters.query.graphql* pasting the code bellow:

```
query GetCharacters {
  characters {
    __typename
    results {
      ...characterData
      image
      species
```

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

Finally, we paste the code bellow at *src/common/graphql/queries/shopping-cart.query.graphql* to create our shopping cart query:

```
query GetShoppingCart {
  shoppingCart @client {
    id
    totalPrice
    numActionFigures
  }
}
```

Updating local cache initialization

Now that we have our local queries, we will update our local cache at *src/common/config/apollo/local-cache.ts*:

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

export const localCache = new InMemoryCache();

export const LocalCacheInitQuery = gql`
  query LocalCacheInit {
    shoppingCart
  }
`;

export function initialLocalCache() {
  localCache.writeQuery({
    query: LocalCacheInitQuery,
    data: {
      shoppingCart: null,
    },
  });
}
```

Here we are initializating our shopping cart with a null value. After that, at *src/common/config/apollo-client.ts* we paste the code bellow to initializate our local cache:

```
import { ApolloClient, ApolloLink } from '@apollo/client';
import { errorLink } from './apollo/error-link';
import { createHttpLink } from './apollo/http-link';
```

```
import { initialLocalCache, localCache } from './apollo/local-cache';

export function createApolloClient() {
  const httpLink = createHttpLink();

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

  return apolloClient;
}

initialLocalCache();
```

Creating our fieldPolicies

Now, instead of using resolvers (they are being deprecated in Apollo v3) we will use the fieldPolicies to initialize the local fields. Back to *src/common/config/apollo/local-cache.ts* we paste:

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

export const localCache = new InMemoryCache({
  typePolicies: {
    Character: {
      fields: {
        chosenQuantity: {
          read(chosenQuantity) {
```

```
            if (chosenQuantity === undefined || chosenQuantity === null) {
              return 0;
            }
            return chosenQuantity;
          },
        },
        unitPrice: {
          read(_, { readField }) {
            const charName = readField('name');
            switch (charName) {
              case 'Albert Einstein':
                return 25;
              case 'Rick Sanchez':
              case 'Morty Smith':
                return 10;

              default:
                return 5;
            }
          },
        },
      },
    },
  },
});

export const LocalCacheInitQuery = gql`
  query LocalCacheInit {
    shoppingCart
  }
`;
```

```
export function initialLocalCache() {
  localCache.writeQuery({
    query: LocalCacheInitQuery,
    data: {
      shoppingCart: null,
    },
  });
}
```

Let's dive in what are we doing here, piece by piece:

- We modify our InMemoryCache object, adding field policies to the Character type.
- At the chosenQuantity's field policy we add a read function based on the existing value of `chosenQuantity` field. If this field is not `falsy` we return its value, if not, return `0`. Since this is a local field, the value obtained from the server is initially falsy.
- At the unitPrice's field policy we add another read function based on the value of name field. To achieve this we use the `readField` function helper, passing the name of the field that we are interested. In this particular case, we are looking for change the character's `unitPrice` based on its own `name`.

Now, at *codegen.yml* we paste:

```
overwrite: true
schema: "https://rickandmortyapi.com/graphql"
documents: "src/**/*.graphql"
generates:
  src/common/generated/graphql.tsx:
    schema: "./src/common/config/local-schema.graphql"
```

```
    plugins:
      - "typescript"
      - "typescript-operations"
      - "typescript-react-apollo"
      - "fragment-matcher"

  src/common/generated/fragment-matcher.json:
    schema: "./src/common/config/local-schema.graphql"
    plugins:
      - "fragment-matcher"
```

And finallly we can generate the typing again running:

```
yarn gen-graphql
```

Updating the character card

After that, we install some dependencies that will be used now, and later on:

```
yarn add react-native-vector-icons @types/react-native-vector-icons @react-navigation/native r
```

and update the *src/common/components/CharacterCard.tsx*:

```
import React from 'react';
import { Image, StyleSheet, Text, View } from 'react-native';
```

```
import { RectButton } from 'react-native-gesture-handler';
import Icon from 'react-native-vector-icons/Entypo';

interface Props {
  data: {
    image?: string | null;
    name?: string | null;
    unitPrice?: number;
    chosenQuantity?: number;
  };
}

const CharacterCard: React.FC<Props> = ({ data }) => {
  return (
    <View style={styles.container}>
      {data.image && (
        <Image source={{ uri: data.image }} style={styles.image} />
      )}
      <View style={styles.details}>
        <Text style={styles.text}>{data.name}</Text>
        <Text style={styles.text}>{`U$ ${data.unitPrice}`}</Text>
      </View>
      <View style={styles.choseQuantityContainer}>
        <RectButton>
          <Icon name="minus" size={24} color="#3D7199" />
        </RectButton>
        <Text style={styles.choseQuantityText}>{data.chosenQuantity}</Text>
        <RectButton>
          <Icon name="plus" size={24} color="#3D7199" />
        </RectButton>
      </View>
    </View>
```

```
    );
  };

  export default CharacterCard;

  const styles = StyleSheet.create({
    container: {
      width: '100%',
      borderRadius: 20,
      marginVertical: 8,
      paddingHorizontal: 8,
      paddingVertical: 24,
      backgroundColor: '#F0F0F0',
      flexDirection: 'row',
    },
    image: { width: 70, height: 70 },
    details: {
      marginLeft: 8,
      justifyContent: 'space-between',
      flex: 1,
    },
    text: {
      fontSize: 16,
      fontWeight: 'bold',
    },
    choseQuantityContainer: {
      flex: 1,
      alignItems: 'center',
      justifyContent: 'space-between',
      flexDirection: 'row',
    },
    choseQuantityText: {
```

```
    padding: 8,
    borderRadius: 8,
    backgroundColor: '#fff',
    fontSize: 16,
    fontWeight: 'bold',
  },
});
```

At this update we are presenting the new local fields `unitPrice`, `chosenQuantity` and the `RectButton` to increase and decrease the quantities. Now we will build this logic to update the `chosenQuantity` at *src/common/hooks/use-update-chosen-quantity.ts*:

```
import { useApolloClient } from '@apollo/client';
import { useCallback } from 'react';
import {
  CharacterDataFragment,
  CharacterDataFragmentDoc,
  GetShoppingCartDocument,
  GetShoppingCartQuery,
} from '../generated/graphql';

interface UpdateChosenQuantity {
  (): {
    onIncreaseChosenQuantity: (id: string) => void;
    onDecreaseChosenQuantity: (id: string) => void;
  };
}

export const useUpdateChosenQuantity: UpdateChosenQuantity = () => {
```

```
  const client = useApolloClient();

  const getCharacter = useCallback(
    (id: string) =>
      client.readFragment<CharacterDataFragment>({
        fragment: CharacterDataFragmentDoc,
        id: `Character:${id}`,
      }),
    [client],
  );

  const getShoppingCartParams = useCallback(() => {
    const shoppingCart = client.readQuery<GetShoppingCartQuery>({
      query: GetShoppingCartDocument,
    })?.shoppingCart;

    if (!shoppingCart) {
      return {
        id: 'ShoppingCart:1',
        totalPrice: 0,
        numActionFigures: 0,
      };
    }

    return {
      ...shoppingCart,
    };
  }, [client]);

  const increaseShoppingCart = useCallback(
    (unitPrice: number) => {
      let { id, totalPrice, numActionFigures } = getShoppingCartParams();
```

```
    totalPrice = totalPrice + unitPrice;
    numActionFigures = numActionFigures + 1;

    client.writeQuery<GetShoppingCartQuery>({
      query: GetShoppingCartDocument,
      data: {
        shoppingCart: {
          id,
          numActionFigures,
          totalPrice,
        },
      },
    });
  },
  [client, getShoppingCartParams],
);

const decreaseShoppingCart = useCallback(
  (unitPrice: number) => {
    let { id, totalPrice, numActionFigures } = getShoppingCartParams();

    totalPrice = totalPrice - unitPrice;
    numActionFigures = numActionFigures - 1;

    if (totalPrice < 0) {
      totalPrice = 0;
    }
    if (numActionFigures < 0) {
      numActionFigures = 0;
    }
```

```
      client.writeQuery<GetShoppingCartQuery>({
        query: GetShoppingCartDocument,
        data: {
          shoppingCart: {
            id,
            numActionFigures,
            totalPrice,
          },
        },
      });
    },
    [client, getShoppingCartParams],
  );

  const onIncreaseChosenQuantity = useCallback(
    (id: string) => {
      const character = getCharacter(id);

      client.writeFragment<CharacterDataFragment>({
        fragment: CharacterDataFragmentDoc,
        id: `Character:${id}`,
        data: {
          ...(character as CharacterDataFragment),
          chosenQuantity: (character?.chosenQuantity ?? 0) + 1,
        },
      });
      increaseShoppingCart(character?.unitPrice as number);
    },
    [client, getCharacter, increaseShoppingCart],
  );

  const onDecreaseChosenQuantity = useCallback(
```

```
    (id: string) => {
      const character = getCharacter(id);

      let chosenQuantity = (character?.chosenQuantity ?? 0) - 1;

      if (chosenQuantity < 0) {
        chosenQuantity = 0;
      }

      client.writeFragment<CharacterDataFragment>({
        fragment: CharacterDataFragmentDoc,
        id: `Character:${id}`,
        data: {
          ...(character as CharacterDataFragment),
          chosenQuantity,
        },
      });
      decreaseShoppingCart(character?.unitPrice as number);
    },
    [client, getCharacter, decreaseShoppingCart],
  );

  return {
    onIncreaseChosenQuantity,
    onDecreaseChosenQuantity,
  };
};
```

Let's carve up what we are doing here:

- Firs't we import our types from the generated file.
- Then we create an interface to type our local api.
- Then we get the `useApolloClient()` hook.
- After that we create a helper `getCharacter` to read our character fragment, passing the fragment doc and the `id` of the fragment (usually the apollo saves the fragments in a normalized way, using the `typename:id` as a unique key).
- After this we create the `getShoppingCartParams` to retrive the `shoppingCart` data from the cache. If the `shoppingCart` is null we return some default values.
- On `increaseShoppingCart` we retrieve the data from `getShoppingCartParams` and add the `unitPrice` from the character being edited. The same happens to *decreaseShoppingCart*.
- On `onIncreaseChosenQuantity` we `getCharacter`, update his chosenQuantity properly and passes its `unitPrice` to the `increaseShoppingCart`. The similar ocurs with the *onDecreaseChosenQuantity*.
- Finally we expose this api.

Finishing the app

Updating the character card

At *src/common/components/character-cart.tsx* we write:

```
import React from 'react';
import { Image, StyleSheet, Text, View } from 'react-native';
import { RectButton } from 'react-native-gesture-handler';
import Icon from 'react-native-vector-icons/Entypo';
import { useUpdateChosenQuantity } from '../hooks/use-update-chosen-quantity';

interface Props {
  data: {
```

```
    id?: string | null;
    image?: string | null;
    name?: string | null;
    unitPrice?: number;
    chosenQuantity?: number;
  };
}

const CharacterCard: React.FC<Props> = ({ data }) => {
  const { onIncreaseChosenQuantity, onDecreaseChosenQuantity } =
    useUpdateChosenQuantity();

  return (
    <View style={styles.container}>
      {data.image && (
        <Image source={{ uri: data.image }} style={styles.image} />
      )}
      <View style={styles.details}>
        <Text style={styles.text}>{data.name}</Text>
        <Text style={styles.text}>{`U$ ${data.unitPrice}`}</Text>
      </View>
      <View style={styles.choseQuantityContainer}>
        <RectButton
          onPress={onDecreaseChosenQuantity.bind(null, data.id as string)}>
          <Icon name="minus" size={24} color="#3D7199" />
        </RectButton>
        <Text style={styles.choseQuantityText}>{data.chosenQuantity}</Text>
        <RectButton
          onPress={onIncreaseChosenQuantity.bind(null, data.id as string)}>
          <Icon name="plus" size={24} color="#3D7199" />
        </RectButton>
      </View>
```

```
    </View>
  );
};

export default CharacterCard;

const styles = StyleSheet.create({
  container: {
    width: '100%',
    borderRadius: 20,
    marginVertical: 8,
    paddingHorizontal: 8,
    paddingVertical: 24,
    backgroundColor: '#F0F0F0',
    flexDirection: 'row',
  },
  image: { width: 70, height: 70 },
  details: {
    marginLeft: 8,
    justifyContent: 'space-between',
    flex: 1,
  },
  text: {
    fontSize: 16,
    fontWeight: 'bold',
  },
  choseQuantityContainer: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'space-between',
    flexDirection: 'row',
  },
```

```
  choseQuantityText: {
    padding: 8,
    borderRadius: 8,
    backgroundColor: '#fff',
    fontSize: 16,
    fontWeight: 'bold',
  },
});
```

Here we are just adding the `onPress` event listener using our local api `useUpdateChosenQuantity`.

Creating the cart screen

Now that we have our shopping cart logic, we can build our cart screen (`src/screens/Cart.tsx`):

```
import React, { useCallback } from 'react';
import { useNavigation } from '@react-navigation/native';
import { StyleSheet, Text, View, SafeAreaView, Button } from 'react-native';
import { useGetShoppingCartQuery } from '../common/generated/graphql';

const Cart = () => {
  const navigation = useNavigation();
  const { data } = useGetShoppingCartQuery();

  const handleNavigation = useCallback(() => {
    navigation.navigate('Home');
  }, [navigation]);

  return (
    <SafeAreaView style={styles.container}>
```

```
      {data?.shoppingCart?.numActionFigures ? (
        <>
          <View style={styles.content}>
            <Text style={styles.emoji}>😁</Text>
            <Text
              style={
                styles.subtitle
              }>{`Total number of items: ${data?.shoppingCart.numActionFigures}`}</Text>
            <Text
              style={
                styles.subtitle
              }>{`Total price: U$ ${data?.shoppingCart.totalPrice}`}</Text>
          </View>
        </>
      ) : (
        <>
          <View style={styles.content}>
            <Text style={styles.emoji}>😢</Text>
            <Text style={styles.title}>Empty cart!</Text>
            <View style={styles.footer}>
              <Button title="Go back to shop" onPress={handleNavigation} />
            </View>
          </View>
        </>
      )}
    </SafeAreaView>
  );
};

export default Cart;

const styles = StyleSheet.create({
```

```
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  content: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
    width: '100%',
  },
  title: {
    fontSize: 24,
    marginTop: 15,
    lineHeight: 32,
    textAlign: 'center',
  },
  subtitle: {
    fontSize: 16,
    lineHeight: 32,
    marginTop: 8,
    textAlign: 'center',
    paddingHorizontal: 20,
  },
  emoji: {
    fontSize: 44,
    textAlign: 'center',
  },
  footer: {
    width: '100%',
    paddingHorizontal: 20,
```

```
  },
});
```

Here we are just adding our Cart view, using the `shoppingCart` query to printing the info. Finally we install the [react-native bottom tabs]:

```
yarn add @react-navigation/bottom-tabs
```

At *src/routes.tsx* we write:

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import { useGetShoppingCartQuery } from './common/generated/graphql';
import MaterialIcons from 'react-native-vector-icons/MaterialIcons';

import Home from './screens/Home';
import Cart from './screens/Cart';

const TabRoutes = createBottomTabNavigator();

const Routes: React.FC = () => {
  const { data } = useGetShoppingCartQuery();

  return (
    <TabRoutes.Navigator
      tabBarOptions={{
```

```
          labelPosition: 'beside-icon',
          style: {
            height: 64,
            alignItems: 'center',
          },
        }}>
        <TabRoutes.Screen
          name="Home"
          component={Home}
          options={{
            tabBarIcon: ({ size, color }) => (
              <MaterialIcons size={size * 1.2} color={color} name="home" />
            ),
          }}
        />
        <TabRoutes.Screen
          name="Cart"
          component={Cart}
          options={{
            tabBarIcon: ({ size, color }) =>
              data?.shoppingCart?.numActionFigures ? (
                <View style={styles.badgeIconView}>
                  <Text style={styles.badge}>
                    {data?.shoppingCart?.numActionFigures}
                  </Text>
                  <MaterialIcons
                    size={size * 1.2}
                    color={color}
                    name="shopping-cart"
                  />
                </View>
              ) : (
```

```
              <MaterialIcons
                size={size * 1.2}
                color={color}
                name="shopping-cart"
              />
            ),
          }}
        />
      </TabRoutes.Navigator>
    );
};
export default Routes;

const styles = StyleSheet.create({
  badgeIconView: {
    position: 'relative',
  },
  badge: {
    position: 'absolute',
    zIndex: 10,
    left: 24,
    bottom: 20,
    padding: 1,
    borderRadius: 20,
    fontSize: 14,
  },
});
```

Then at *src/index.tsx* we finally update:

```
import React from 'react';
import { ApolloProvider } from '@apollo/client';
import { NavigationContainer } from '@react-navigation/native';
import { createApolloClient } from './common/config/apollo-client';

import Routes from './routes';

const apolloClient = createApolloClient();

const App = () => {
  return (
    <ApolloProvider client={apolloClient}>
      <NavigationContainer>
        <Routes />
      </NavigationContainer>
    </ApolloProvider>
  );
};

export default App;
```

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 cart screen with the total price and the total number of action figures in the shopping cart. Finally, I'll be happy if you could provide me any feedback about the code, structure, doubt or anything that could make me a better developer!

Creating an React Native app using Apollo Graphql v3 (2 Part Series)

1. **Creating a React Native app using Apollo Graphql v3**
2. Unit tests with react-native testing library and Apollo Grap...

Discussion (0)

Subscribe

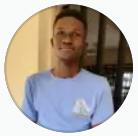
DEV

Add to the discussion

Read next

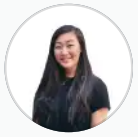
5 Reasons Why You Should Never, Ever Write Tests

Kai - Aug 30



CSS Tutorial – Build a landing page using just HTML and CSS featuring Flexbox and Grid.

Kingsley Ubah - Sep 2



Building Conway's Game of Life in Javascript

Saji Wang - Sep 2

Learn Full Stack Javscript Online

Javascript_Eduonix - Sep 3

Harrison Henri dos Santos Nascimento

Follow

EDUCATION

Eletronical Engineering

WORK

Junior Developer at Alelo S.A

JOINED

Jun 19, 2021

More from Harrison Henri dos Santos Nascimento

Unit tests with react-native testing library and Apollo Graphql

#reactnative #testing #tutorial