Querying GraphQL in React

This guide serves as a starting point for you to work with querying GraphQL using Apollo in React. many of the interactions between these frameworks are difficult to understand, and many of the interactions are poorly documented. This guide aims to fix that by breaking down the types of queries you will see

GraphQL Fragments

GraphQL fragments are ways to easily use parts of GraphQL code to make your queries shorter and easier. Below is an example fragment in Javascript. Note that the actual fragment is in gql synatx, I chose to show the fragment in Javascript to help avoid confusion in our implementation.

Fragment Example

```
export const agentInterface = gql`
fragment agentInterface on Agent {
  id
  name
  type
  image
  note
  ownedEconomicResources{
    id
  }
  agentProcesses{
    id
  }
  agentPlans{
    id
  }
  agentEconomicEvents{
    id
  }
  agentCommitments{
    id
  }
  agentRelationships{
    id
  }
  agentRoles{
    id
  }
  agentRecipes{
    id
  }
}`;
```

As you can see from the example above, the fragment starts with

Fragment Declaration

```
fragment myName on ObjectName
```

This is then followed by the attributes of your object that you want to return from the fragment.

Fragment Attributes

```
id
  name
  type
  image
  note
```

There are also other object types shown below the attributes of agent. these can be opened with braces{} and any attributes you want from those objects can be included within the brackets, including other objects.

Fragment Objects

```
SomeObject{
 Attribute
 SomeOtherObject{
  AttributeOfThisObeject
  YouCanDoThisAsLongAsYouWant{
   ObjectsAllTheWayDown
  }
 }
}
```

ViewerQueries

Queries serve as a way to get the object from the database. All of the queries available to you can be seen in the ViewerQuery section of the HTML documentation that is generated in Valuenetwork API Documentation. These Queries will be broken down into their three seperate parts; the QraphQL query, The Apollo linking, and the React UI.

Non-Parameterized Queries

Non-Parameterized queries are queries which require no parameters to run (like "this()" and not like "notThis(int ThisIsTheWrongKind)")

GraphQL

Below is the GraphQL portion of the query for the Unit Object.

GraphQL

```
const query = gql`
query($token: String) {
  viewer(token: $token) {
    allUnits{
      ...unitInterface
    }
  }
}
${unitInterface}
`;
```

The 'Query' portion denotes it as a GraphQL query

The 'Viewer' is exposed by the backend. this is how you interact with it

'allUnits' is the query you are calling

'...unitInterface' inserts all of the attributes of the Unit Interface into the query to be queried. Note how the interface needs to be passed in on the bottom

Apollo

Below is the Apollo portion of the query for the Unit Object.

Apollo

```
export default compose(
  // bind input data from the store
  connect((state: AppState) => ({
    variables: {
      token: getActiveLoginToken(state),
    },
  })),

  // Make the query
  graphql(query, {
    // read query vars into query from input data above
    options: (props) => ({ variables: {
      ...props.variables,
    } }),

    // transform output data
    props: ({ ownProps, data: { viewer, loading, error, refetch } }) =>
(
      {
        loading,
        error,
        refetchAgent: refetch,  // :NOTE: call this in the component to
force reload the data
        unitList: viewer ? viewer.allUnits : null,
      }),
  })
)
```

For this section, the comments provided in the code describe the function of this method well.

The important part is the transformation of the output data. here you can structure how we pas the object back to the front end

React

The core React for the query is in two parts. The display of the object(Note: concatArray is a helper function we used to display an array of objects as a string of their IDs)

Object

```
export const Agent = (props) => {
  let agent = props.agent;
  return(
    <div>
      <div>id: {agent.id}</div>
      <div>name: {agent.name}</div>
      <div>type: {agent.type}</div>
      <div>image: {agent.image}</div>
      <div>note: {agent.note}</div>
      <div>ownedEconomicResources:
{concatArray(agent.ownedEconomicResources)}</div>
      <div>agentProcesses: {concatArray(agent.agentProcesses)}</div>
      <div>agentPlans: {concatArray(agent.agentPlans)}</div>
      <div>agentEconomicEvents:
{concatArray(agent.agentEconomicEvents)}</div>
      <div>agentCommitments: {concatArray(agent.agentCommitments)}</div>
      <div>agentRelationships:
{concatArray(agent.agentRelationships)}</div>
      <div>agentRoles: {concatArray(agent.agentRoles)}</div>
      <div>agentRecipess: {concatArray(agent.agentRecipes)}</div>
      <br/>
    </div>
  );
};
```

And the call to Apollo

Call to Apollo

```
export const GetAllAgents = getAllAgents(({ agent, loading, error}) => {
  if (loading) {
    return(
      <strong>Loading...</strong>
    );
  } else if (error) {
    return(
      <p style={{color: "#F00"}}>API error</p>
    );
  } else {
    return(
      <div>
        {agent.map( (singleAgent) => (<Agent key={singleAgent.id}
agent={singleAgent}/>))}
      </div>
    );
  }
});
```

The call displays text in the event of an error, or, under normal operation, displays the React code for an agent object for each individual agent returned by the query

Parameterized Queries

Paramaterized queries are queries that take in an argument as input. they are useful for getting a single object by its id

GraphQL

Below is the GraphQL portion of the query for the Agent Object.

GraphQL

```
const query = gql`
query($token: String, $AgentId: Int) {
  viewer(token: $token) {
    agent(id: $AgentId){
      ...agentInterface
    }
  }
}
${agentInterface}
`;
```

The 'Query' portion denotes it as a GraphQL query. You need to pass your parameters into the query here.

The 'Viewer' is exposed by the backend. this is how you interact with it

'agent' is the query you are calling. Here you can pass along the attributes required for the query

'...agentInterface' inserts all of the attributes of the Agent Interface into the query to be queried. Note how the interface needs to be passed in on the bottom

Apollo

Below is the Apollo portion of the query for the Agent Object.

Apollo

```
export default compose(
  // bind input data from the store
  connect((state: AppState) => ({
    variables: {
      token: getActiveLoginToken(state),
    },
  })),
  graphql(query, {
    // read query vars into query from input data above
    options: (props) => (
      {
        variables: {
          ...props.variables,
          AgentId: props.agentId
        }
      }),
    // transform output data
    props: ({ ownProps, data: { viewer, loading, error } }) => (
      {
        loading,
        error,
        agent: viewer ? viewer.agent : null,
      }),
  })
);
```

The first portion of this binds the connection. Then, in the 'graphql' section:

The first part("options") sets up the query. The parameters need to be passed in here.

The second part("props ") returns the object form the query.

React

The core React for the query is in three parts. The display of the object(Note: concatArray is a helper function we used to display an array of objects as a string of their IDs)

Object

```
export const Agent = (props) => {
  let agent = props.agent;
  return(
    <div>
      <div>id: {agent.id}</div>
      <div>name: {agent.name}</div>
      <div>type: {agent.type}</div>
      <div>image: {agent.image}</div>
      <div>note: {agent.note}</div>
      <div>ownedEconomicResources:
{concatArray(agent.ownedEconomicResources)}</div>
      <div>agentProcesses: {concatArray(agent.agentProcesses)}</div>
      <div>agentPlans: {concatArray(agent.agentPlans)}</div>
      <div>agentEconomicEvents:
{concatArray(agent.agentEconomicEvents)}</div>
      <div>agentCommitments: {concatArray(agent.agentCommitments)}</div>
      <div>agentRelationships:
{concatArray(agent.agentRelationships)}</div>
      <div>agentRoles: {concatArray(agent.agentRoles)}</div>
      <div>agentRecipess: {concatArray(agent.agentRecipes)}</div>
      <br/>
    </div>
  );
};
```

The call to Apollo

Call to Apollo

```
export const GetSingleAgent = getAgentById(({ agent, loading, error })
=> {
  if (loading) {
    return(
      <strong>Loading...</strong>
    );
  } else if (error) {
    return(
      <p style={{color: "#F00"}}>API error</p>
    );
  } else {
    return(
      <div>
        <Agent agent={agent}/>
      </div>
    );
  }
});
```

And the call in the main page body

Main Body

```
<AgentField setAgent={this.getAgentById}
onSubmitAction={this.stopRefresh}/>
{getOneAgentId ? <GetSingleAgent agentId={getOneAgentId}/> : <div>Enter
a value</div>}
```

The first line is a button and text field which lets the user select which ID they want to see.

The second line checks that an ID has been asked for, and if it has, calls Apollo, passing in the selected agentID