**GraphQL with React - The Complete Developers Guide**

**Section 1: Why GraphQL**

1. GraphQL aims to solve the issue of querying highly relational API data

**Section 3: On To GraphQL**

1. The typical GraphQL architecture is as follows:
   1. **Datastore**
      1. This is the database which holds the data (SQL or no-SQL) to be queried by GraphQL
   2. **Express/GraphQL Server**
      1. The back-end server which holds the GraphQL logic used to query data
      2. Typically, an Express-like server
   3. **Graph*i*QL Endpoint**
      1. The graphical user interface which exposes the GraphQL data to be queried using the GraphQL engine and specific query language
      2. Graph*i*QL is a setting on the server that can be turned on and off i.e., on for dev, but off for production
2. Creating an Express-based GraphQL server requires 3 npm packages:
   1. **Express**
      1. A backend HTTP(S) API server building library
   2. **express-graphql**
      1. An integration library (compatibility layer) between express and graphql
   3. **graphql**
      1. The core GraphQL library
3. To initiate a GraphQL server:
   1. In a terminal, run the commands:
      1. **npm init** to initialize a new npm application
         1. Take all the default options if there is no customizations
      2. npm install express express-graphql graphql to install all the packages
   2. In a code editor:
      1. Create a **server.js** file in the root directory to initiate the GraphQL server
         1. Import the **express** package
         2. Create a const called **app** assigning it the called **express()** method
         3. Call **app.listen()** on the desired port (first parameter) to start the express app at runtime
         4. Express will be configured to either handle a regular HTTP(S) web request or to process a GraphQL query
            1. GraphQL can just be relegated to one portion of the express application, it does not have to occupy the entire express server
         5. \*Import the **graphqlHTTP** variable from the **express-graphql** library
            1. The variable can be declared as **expressGraphQL**
         6. Call the **app.use()** function to add GraphQL as a middleware service to handle incoming GraphQL query requests
            1. Pass in the string **‘/graphql’** as the first, path, parameter

This tells the express application that all requests going to route ‘/graphql’ should be handled by the proceeding engine

* + - * 1. Pass a call to the **expressGraphQL()** method in as the second parameter

Pass in a JSON object as the **options** (of type **OptionsData** declared in the express-graphql module) parameter with the properties:

**graphiql** set to **true** to enable the developer-friendly graphiql user interface on the GraphQL server

Pass in a GraphQL **schema** as the second parameter

* + 1. Create a **schema** folder
       1. Create a file called **schema.js** inside of the **schema** folder where the **schema** of the GraphQL data to be queried is defined

1. The purpose of GraphQL **Schema** is to instruct GraphQL as to what type of data exists in the application
2. To define a schema:
   1. Import the core **graphql** module
   2. Destruct the **GraphQLObjectType** type from the imported **graphql** variable
      1. This object type is used to define a GraphQL object
   3. Declare a constant called **[Object]Type**, where [Object] is the name of the schema object to be queried by GraphQL
      1. Assign this schema object to a **new GraphQLObjectType()** instance
         1. Pass in a JSON object as the **config** parameter (of type **GraphQLObjectTypeConfig**)
            1. Define the **name** property to provide the name of the object being defined
            2. Define a **description** object and assign it to a short description about the object

This will only be seen by developers during development

* + - * 1. Define a **fields** property (of type **GraphQLFieldConfigMap**) to describe the different properties of the schema object being defined

The **keys** of this object are the names of the properties declared on the defining object

The values of this object (of type **GraphQLFieldConfig**) describe what type of data each property is

Assign an object to each value containing:

A **type** property to declare the field’s type

Assign this property with a GraphQL ObjectType which can be destructed from the imported graphql module

**\***This property can be assigned to any GraphQLObjectType; including default GraphQL types AND custom schema-specific GraphQLObjectTypes defined above it

**\*\***The **fields** property SHOULD be assigned to an arrow function (() => {}) to allow for typing objects using circular references

This allows for fields to have a GraphQLObjectType of an object defined later in the schema file

**\***If circular reference is defined in a fields property that is NOT enclosed in an arrow function, an error message : “**ReferenceError: [type] is not defined**” will result, where [type] is the GraphQLObjectType that is circularly referenced

* 1. The Imported graphql module has several **GraphQL Object Types** that can be destructed from it:
     1. The **GraphQLString** type represents a string
     2. The **GraphQLInt** type represents an integer
  2. \*Define a **Root Query** object as an entry point into the GraphQL Schema:
     1. Its purpose is to allow GraphQL to access any node in the graph of data defined by the schema
     2. Declare a const named **RootQuery** and assign it to a new instance of type **GraphQLObjectType**
        1. Pass a **GraphQLObjectTypeConfig** object into the constructor with the properties:
           1. Assign **‘RootQueryType’** to the name property
           2. Assign the description property to a short message stating this is the root query
           3. Assign the fields property to a **GraphQLFieldConfigMap** list which describes the Root’s accessible nodes, their types, and the query arguments they accept:

Each property declared in this list should have the name of the (application-specific) node which GraphQL can access during a query

Assign each property with a **GraphQLFieldConfig** object containing:

A **type** property assigned to an instantiated GraphQLObjectType object, representing an accessible node GraphQL can query

I.e. UserType, ProductType, CustomerType

This describes the return type resulting from a query to this node

An **args** property assigned to a **GraphQLArgumentConfigMap** object which represents a key-value list of the arguments a specific query to this node accepts (keys), along with its **GraphQLArgumentConfig** object values

Set the **type** property of this value to the GraphQL type which describes the object

A **resolve()** function (as type **GraphQLFieldResolver**) which actually executes the query against this node and returns the data being queried

Pass in a **parentValue** (**source**) parameter to be used in this function to access properties of the object being queried

Pass in an **args** parameter to be used in this function to retrieve the query arguments (as properties on this parameter) defined in the above GraphQLArgumentConfig property

This method can return a Promise<> for asynchronous data fetching

\*This is the most important part of the GraphQL schema because it actually performs a query against a data store and returns data

* 1. Destruct the **GraphQLSchema** object from the imported graphql module
  2. Instantiate a **new GraphQLSchema** instance and pass in a **GraphQLSchemaConfig** object to its constructor which contains:
     1. A **query** property and assign it to the **RootQuery** object defined above
  3. Export the GraphQLSchema object defined above

1. In the server.js file:
   1. Import the **schema** object defined in the above schema**.**js file
   2. Pass the schema as an argument to the second property to OptionsData object passed into the expressGraphQL method
2. Open the URL localhost:[listeningPort]/gaphql, where [listeningPort] is the port assigned to listen to incoming web requests in the express-graphql server (i.e. 4000), to interact with Graph*i*QL user interface once a Graph*i*QL enabled server has been stood up
3. The GraphQL query language is **NOT** written in JavaScript; GraphQL has its own unique query language
4. **\*\*\***Use the **CTRL+/** hotkeys to comment/uncomment lines of code in Graph*i*QL
5. The default Query Type is **query**
   1. To write a query to fetch a single entity use the following syntax:

**[queryType]? [queryName]?($[var1]: [varType1]!,…) {**

**[alias]?: [entity]([param1]: "[value1]", [param2]: “[value2]”) {**

**[field1],**

**[field2],**

**[field3]**

**}**

**}**

* + 1. Where **[queryType]?** is the type of query being executed
       1. Optional, defaulted to **query**
    2. Where **[queryName]?** is the name of the query itself
       1. Optional, not necessary to perform a query
       2. Naming a query is helpful when building client-side queries
    3. Where **[varX]** is a reference to a query variable passed into the query
       1. **\***MUST be preceded with a ‘**$**’
    4. Where **[varTypeX]** states the GraphQL type that [varX] is
       1. **\*\***Place a ‘**!**’ after [varTypeX] to state it’s required
    5. Where **[alias]?** is the alias name assigned to the [entity] being queried
       1. Optional, defaults to [entity] name
    6. Where **[entity]** is a key string of the GraphQLFieldConfigMap object assigned to the Root Query’s **fields** property:
       1. Where **[paramX]** is query parameter; a key string of the GraphQLArgumentConfigMap object assigned to the **args** property of a GraphQLFieldConfig object assigned to a fields property in the Root Query Object
          1. If the **()** in the query contains one or more incorrect parameters (any argument not specified in an [entity]’s args property), the query will return the error:

**"Syntax Error: Expected Name, found \")\"."**

* + - 1. Where **[valueX]** is the value of each query parameter
      2. **\***The type of [entity] is inferred (based on the value assigned to its **type** field in the Root Query)
         1. **\***Type coercion is handled behind the scenes GraphQL and does not need to be programed
    1. Where **[fieldX]** is a key string of the GraphQLFieldConfigMap assigned to the **fields** property of a GraphQLObjectType instance referenced by **type** (assigned to **type** property of [entity]’s GraphQLFieldConfig object) in the Root query
       1. **\***Each [fieldX] name is optional to the query
          1. If [fieldX] is not present in the query than it will NOT be returned in the result data
          2. This behavior helps to limit the amount of data sent back (stored) to the client device (i.e., a cell phone with low-bandwidth signal and limited memory)
  1. Queries are made against the Root Query type

1. Query results of a GraphQL query are returned in the format:

**{**

**"data": {**

**"[entity]": {**

**"[field1]": "[value1]",**

**"[field2]": "[value2]",**

**"[field3]": "[value3]",**

**}**

**}**

**}**

* 1. Where the variables are the same as listed queries above
  2. **\*\*“data”** is always the top-level property
  3. The resulting object(s) assigned to “data” are the values returned from the resolve() functions assigned to each entity being queried
  4. The results are returned in **JSON** format
  5. **\***If no [entity] that fulfills the specified query parameters exists, then a value of **null** is assigned to the entity name in the results set

1. A GraphQL server can be used as an abstraction layer to access data from multiple servers connected to multiple databases
2. Use the **json-server** npm library to stand up a dummy no-SQL JSON data server

**Section 4: Fetching Data with Queries**

1. Once a GraphQLObjectType is defined in GraphQL, any declared fields on this object can be returned from a query to this node
2. To allow for a **Nested Query** in GraphQL:
   1. **\***Place the type definition of the nested object ABOVE the instance of the GraphQLObjectType that it will be nested in
   2. The association between one GraphQLObjectType definition and another is placed on the type definition that contains the embedded type
      1. The associating field is declared like any other GraphQLObjectType field
      2. Place a field named **[entity]**, where [entity] is the name of the field binding the association, in the fields property(ies) of the parent GraphQLObjectType fields definition
         1. Assign the **type** of this field to the associated [entity]Type defined above it
         2. Declare a **resolve()** function:
            1. Pass a **parentValue** object as the first parameter

This parameter is an object of the GraphQLObjectType that the parent object is defined as (i.e. User)

* + - * 1. In the function’s body, use the associating id property of the parentValue object (i.e. parentValue.companyId) to query the resulting object by
      1. **\***The **resolve()** function is required

1. To graph a one-to-many relationship in GraphQL:
   1. Import/destruct the **GraphQLList** object type from the graphql module
   2. In the fields property of the parent GraphQLObjectType constructor:
      1. Declare a property with the name of the child node (i.e. users)
         1. Assign this property to a GraphQLFieldConfig object with its **type** value equal to a **new GraphQLList()** instance
            1. Pass in the GraphQLObjectType of the child object being associated
      2. Declare a resolve() function:
         1. In the function’s body, Use the parentValue parameter of this function to query a list of child objects by the parent object’s id
2. If multiple nodes are queried using the same [entity] name, where [entity] is the name of a node in the graph:
   1. Use an assign an alias to the [entity] being queried (as stated in the single entity query syntax above)
3. Use **Query Fragments** to create a shorthand syntax for returning commonly used fields of defined GraphQL node
   1. Use the syntax :

**fragment [fragmentName] on [entity] {**

**[field1]**

**[field2]**

**[field3]**

**}**

* 1. Where **[fragmentName]** is the name of the fragment to reference in a query
  2. Where **[entity]** is a node object defined GraphQL graph
  3. Where **[fieldX]** is the name of a field property defined on the [entity]

1. **Mutations** are used to change data in GraphQL
   1. They can be used to create, update, or delete records in GraphQL nodes
   2. Mutations are made inside of a GraphQL **Root Mutation** node that sits at the root of the GraphQL schema “beside” the Root Query node
2. Wrap a GraphQLObjectType value, when assigning it to a type property, in a new instance of the **graphQLNonNull** object to declare it as non-nullable
   1. i.e.: firstname: { type: **new GraphQLNonNull(**GraphQLString**)** }
3. To create a GraphQL **mutation** object:
   1. Create a const **mutation** variable and assign it to a new instance of the GraphQLObjectType
      1. In the GraphQLObjectTypeConfig object passed to the constructor:
         1. Assign the string: **‘Mutation’** to its name property
         2. Assign a brief description to its description property
         3. Assign the fields property to a **GraphQLFieldConfigMap** list which describes the Root Mutation’s accessible mutation actions (nodes), their types, and the query arguments they accept:
            1. Each property declared in this list should have the name of the mutation action node which GraphQL can access during a query

Assign each property with a GraphQLFieldConfig object containing:

**\***A **type** property assigned to an instantiated GraphQLObjectType object, representing **\***the type of data returned from its resolve function

I.e. UserType, ProductType, CustomerType

**\***Sometimes, the type of data being mutated, and the type of data being returned are not the same

An **args** property assigned to a GraphQLArgumentConfigMap object which represents a key-value list of the arguments a specific mutation query accepts/needs to perform its mutation (keys), along with its **GraphQLArgumentConfig** object values

Set the **type** property of this value to the GraphQL type which describes the object

A **resolve()** function (as type **GraphQLFieldResolver**) which actually executes the mutation against live data and returns an object(s) that results from the mutation

Pass in a **parentValue** (**source**) parameter to be used in this function to access properties of the object being mutated

Pass in an **args** parameter to be used in this function to retrieve the mutation arguments (as properties on this parameter) defined in the above GraphQLArgumentConfig property

This method can return a Promise<> for asynchronous data fetching

* 1. In the constructor of the instantiated GraphQLSchema object being exported:
     1. Add a **mutation** property to the passed in GraphQLSchemaConfig object and assign it to the Root Mutation object

1. **\***When reading a GraphQL query (i.e., in Graph*i*QL), an exclamation mark (!) after an objects type means that the query parameter is required
   1. i.e.: addUser(firstName: String**!**, age: Int**!**, companyId: String)
   2. In the example above, firstName and age are both required
2. **\*\***When writing a GraphQL mutation, one is required to return properties from the mutation’s return type in the mutation query

**Section 6: Clientside GraphQL**

1. Use **MongoLab** as a good MongoDB cloud server
2. The hierarchy of an Apollo Client application is as follows:
   1. **Apollo Provider**
      1. Takes data from the Apollo Store and injects it into the react application
      2. Is the glue layer between Apollo Store and the react application
      3. Contains the vast majority of setup code
   2. **React application**
   3. **Apollo Store**
      1. An abstract piece of technology that is agnostic to the client-side framework (react app)
      2. A store of data that communicates with the GraphQL server and stores data that is retrieved from it
   4. **GraphQL Server**
3. To configure a react application to use GraphQL via Apollo:
   1. Navigate to the index.js (or whichever file instantiates the root element of a react app)
      1. Import the **ApolloClient** **, ApolloProvider,** and **InMemoryCache** modules from the **apollo-client** library
      2. Create a const variable and assign it to a **new ApolloClient()** instance
         1. Pass in an object representing the **ApolloClientOptions** as a constructor parameter
            1. Declare a property named **uri** and assign it to the path where the GraphQL server is located

I.e. **“/graphql”**

\*\*The uri property MUST be defined, as ApolloClient will NOT know where to send its queries to without it

* + - * 1. Declare a property named **cache** and assign it to a **new** **InMemoryCache()** instance

**\***The cache property is required by the ApolloClient object

* + 1. Wrap the entire root element of the application with the **<ApolloProvider>** tag
       1. Assign the variable holding the new ApolloClient (**{client}**) to the **client** prop of the <ApolloProvider> tag

1. To write a GraphQL query, using a GraphQL client library (Apollo):
   1. Import the **gql** and **useQuery** modules from the **@apollo/client** library
   2. Assign a **gql** query to a const variable using the syntax:

**const [VAR\_NAME] = gql`**

**[graphQLQuery]**

**`**

* + 1. Where **[VAR\_NAME]** is the name of the variable being assigned the query
       1. **\***Standard syntax is all caps with **\_** separating words
    2. Where **[graphQLQuery]** is the query itself written in GraphQL query syntax
       1. **\*\***It is important to provide a query name inside the back-ticks, as this name will be viewable in the browser’s **Apollo Extension**
    3. The back-ticks (**``**) are required as gql interoperates its queries as string templates
  1. Use the Graph*i*QL tool to test the [graphQLQuery] before using it

1. **\***A GraphQL query is NOT written in valid JavaScript code
2. To write a react functional component that queries data using the **@apollo/client** library:
   1. Create a functional component file in a react project:
      1. Import the **gql** and **useQuery** modules from the @apollo/clientlibrary
      2. Create a gql query as shown above
      3. Inside the functional component:
         1. Destruct the **loading**, **error**, and **data** properties from the object returned from a call to the **useQuery()** method
            1. Pass the name of the GraphQL query as a parameter of the useQuery() method
            2. The **loading** property is set asynchronously by GraphQL; **true** when loading and **false** when not
            3. The **error** property is set asynchronously by GraphQL

If an error occurs during the query, it will NOT be **null**

Use its **message** property to view the error message (if any)

* + - * 1. The **data** property is set asynchronously by GraphQL

Results of the query will populate in this variable

* + - * 1. **\*\***If the GraphQL query requires variables:

Pass an object (of type **QueryHookOptions**) containing a **variables** key and a value equal to another object with key-value pairs of all the query parameters to the **options** parameter

* + - 1. Call React’s **useEffect()** method to asynchronously change the component as the query executes or errors out
         1. Pass an effect method as the first parameter
         2. Pass an array with useQuery()’s loading, error, and/or data properties as the effect’s dependencies as the second parameter
         3. **\*\*\***The destructed useQuery variables MUST be declared BEFORE the useEffect hook is called in order to use them in the useEffect()
      2. Use the destructed data variable in JSX of the react component
         1. \*\*ALL query results will populate under the root **data** object
  1. Example:

**import React from 'react';**

**import {useParams} from 'react-router-dom';**

**import {gql, useQuery} from '@apollo/client';**

**const GET\_SONG = gql`**

**query GetSong($id: ID!) {**

**song(id: $id) {**

**id**

**title**

**}**

**}`;**

**const SongDetail = () => {**

**const {id} = useParams();**

**const {loading, error, data} = useQuery(GET\_SONG, {variables: {id}});**

**if (loading) return <p>Loading...</p>;**

**if (error) return <p>Error: {error.message}</p>;**

**const {song} = data;**

**return (**

**<div>**

**<h3>{song.title}</h3>**

**</div>**

**);**

**}**

**export default SongDetail;**

**Section 7: Gotchas with Queries in React**

1. **\***When using **react-router-dom** in react, make sure to wrap the **<Router/>** component with the <ApolloProvider/> tag
2. To add **Query Variables** to pass into a GraphQL mutation:
   1. Use the syntax:

**mutation** **[mutationFunctionName****]($[queryVarName]: [queryVarType]…) {**

**[mutationName]([mutationParamName]: $[** **queryVarName]…) {**

**[field1]**

**[field2]**

**}**

* + 1. Where **[mutationFunctionName]** is the name of the arbitrary (developer given) name of the mutation being executed
    2. Where **[queryVarName]** is the name of the query variable; a parameter to be passed into the mutation
       1. **\***MUST be prefixed with a dollar sign (**$**)
    3. Where **[queryVarType]** is the GraphQL type of the query variable (as defined in the GraphQL schema)
    4. Where **[mutationName]** is the name of the GraphQL mutation (as defined in the GraphQL schema)
    5. Where **[mutationParamName]** is the name of the argument to be passed into the mutation
       1. If using query variables, this should coincide with a [queryVarName] passed into the parent mutation function
       2. **\***[queryVarName] MUST be prefixed with a dollar sign (**$**)
    6. Where **[fieldX]** represents a field defined on the node being mutated
    7. Mutations may have multiple query variables
  1. Query Variables assist in filtering and pagination
  2. Example:

**mutation AddSong($title: String) {**

**addSong(title: $title) {**

**id**

**title**

**}**

**}**

* 1. To add query variables to a mutation using the Graph*i*QL interface:
     1. Go to the **Query Variables Pane** (located underneath the Query pane)
        1. Write curly brackets, **{},** to envelope each variable
        2. List variables in proper JSON syntax within the curly brackets
           1. **\***Be sure to wrap every field name with double quotes, **“”**
     2. \*If a query variable is listed yet NOT used in a Graph*i*QL query, the interface will place a red squiggly line under the variable informing the user that it has not been used
     3. Example:

**{**

**“title”: “Sprite vs Coke”**

**}**

* 1. A query variable of type **ID** represents a no-sql like ID in GraphQL

1. To mutate data in a GraphQL query within a react application:
   1. Import the **useMutation** function form the @apollo/client library
   2. Call the useMutation function using the syntax:
      1. **const [[mutateFunc], { data, loading, error }] = useMutation([gqlVar]);**
         1. Where **[gqlVar]** is a GraphQL mutation variable returned from a call to the @apollo/client gql function
         2. Where **[mutateFunc]** is the reference function to call to perform the mutation declared by the [gqlVar]
            1. A call to this returns a promise which can be awaited or appended with **.then()**
            2. Where **error** (string) is any error returned from a failed query
            3. Where **loading** is Boolean that indicates whether the data form the query is loading or not
            4. Where **data** is the actual data that is returned from the query

Each resulting object from the query is wrapped in this data object

* 1. During an event where the mutation (wrapped in [mutateFunc]) is to actually be performed:
     1. Call [mutateFunc]
        1. Pass in an object (of type **MutationFunctionOptions**) as its **options** parameter
           1. Assign this object’s **variables** property to an object containing the key-value pairs of all the query variables to be passed into the GraphQL mutation
  2. Example:

**import React, {useState} from 'react';**

**import {gql, useMutation} from '@apollo/client';**

**const ADD\_SONG = gql`**

**mutation AddSong($title: String) {**

**addSong(title: $title) {**

**title**

**}**

**}**

**`;**

**const SongCreate = () => {**

**const [title, setTitle] = useState('');**

**const [addSong, { data, loading, error }] = useMutation(ADD\_SONG);**

**if (loading) return <div>Submitting Query</div>;**

**if (error) return <div>Error: {error.message}</div>;**

**const onSubmit = (evt) => {**

**evt.preventDefault();**

**addSong({variables: { title }})**

**.then(data => { console.log("Mutation Data: ", data) });**

**}**

**return (**

**<div>**

**<h3>Create a New Song</h3>**

**<form onSubmit={onSubmit}>**

**<label>Song Title:</label>**

**<input onChange={event => setTitle(event.target.value)} value={title}/>**

**</form>**

**</div>**

**);**

**}**

**export default SongCreate;**

1. Use the **refetchQueries** option of the useMutation() function to enable the apollo client to re-query (refresh) data after a mutation has taken place
   1. Use the syntax:

const [[mutateFunc], { data, loading, error }] = useMutation([gqlVar], **{**

**refetchQueries: [{query: [GQL\_QUERY], variables: [vars]}]**

**}**);

* + 1. Where **[GQL\_QUERY]** is the name of a constant variable that is assigned a GraphQL query returned from a call to the @apollo/client gql function
    2. Where **[vars]** is an object containing key-value properties of any query variables that need to be passed into the [GQL\_QUERY] query
  1. **\*\***It is good practice to place all gql queries inside of a separate file (i.e. queries.js) so that they may be accessed within all components that need to query or mutate GraphQL data
  2. **\*\***The caveat of this strategy is that calling the refetchQueries option incurs the expense of another network call to the GraphQL database to retrieve the data returned by [GQL\_QUERY]
  3. When using refetchQueries, GraphQL will NOT call the same query multiple times in react if the query is also loaded in another component
     1. Unless the query is loaded again by application code somehow