Course Overview

Course Overview

Hi everyone, my name is Adhithi Ravichandran, and welcome to my course, Consuming a GraphQL API with Apollo Client and React. I am a software consultant, author, speaker, and blogger. You can follow me on Twitter at @AdhithiRavi or visit my website adhithiravichandran.com to keep in touch and learn more from me. Did you know that with GraphQL the client has more power to ask for what they want and get exactly that in a single call? In this course, we're going to learn to consume and interact with GraphQL APIs from a client React application. We'll be using Apollo Client to consume the GraphQL APIs. Some of the major topics I will cover include setting up our React and Apollo Client dev environment. Then we'll be exploring the GraphQL schema with introspection queries. From there, we'll be writing queries to retrieve GraphQL data and updating our UI. And finally, we'll perform mutations to modify and update GraphQL data. By the end of this course, you'll have the skills and knowledge to build front‑end applications that can consume GraphQL API using the Apollo Client. Before beginning the course, you should be familiar with the fundamental concepts of GraphQL and JavaScript. I hope you'll join me on this journey to learn GraphQL with the Consuming a GraphQL API with Apollo Client and React course on Pluralsight.

Setup React and Apollo Development Environment

Version Check

Introduction

Hello. Welcome to my course on Consuming a GraphQL API with Apollo Client and React. Before we get started with this course, there are two prerequisites that I would like you to have. The first prerequisite is knowledge of GraphQL basics. I'm going to assume that you're aware of the basic concepts of GraphQL and what it is being used for. If you're absolutely new to GraphQL, you can take the first course on this path, GraphQL: The Big Picture. This course gives you a big picture overview of GraphQL, its core concepts, along with examples. You'll be at a good spot with GraphQL prerequisites after completing this course. The next prerequisite for this course is JavaScript. We're going to use a JavaScript application in this course to consume the GraphQL API. There are plenty of courses on JavaScript on Pluralsight that you can explore to get proficient at it. Alright, with the prerequisites out of the way, let's look into what this course is going to teach. In this course, we'll first learn what the Apollo Client is and why we're using it. We'll then set up our React and Apollo Client dev environment. Next, we'll learn to write queries against our GraphQL API to retrieve data. And finally, write mutations to modify or update data. By the end of this course, you're all set to integrate Apollo Client within your React app and consume a GraphQL API for your data needs. In the first module, we're going to work on getting set up with our React and Apollo dev environment. We'll first learn about what is Apollo Client. Then I'll give you a tour of the Globomantics conference app that we're going to be using throughout this course. After that, we'll be all hands‑on to get started with the installation and setup. During the setup, we'll integrate and configure the Apollo Client within the Globomantics React app. We'll then set up our VS Code extensions on the IDE, and I'll walk you through the steps to connect our server schema that we already have available. Finally, we'll explore the server schema on the GraphiQL Playground. I hope you're all set, and let's get started on our journey towards consuming a GraphQL API from our client.

Apollo Client

In this section, let's learn about what the Apollo Client is. Let's take a step back for a second and revisit the definition of GraphQL. GraphQL is a query language for APIs. It gives clients the power to ask for exactly what they need and nothing more, it makes it easier to evolve APIs over time, and enables powerful developer tools. The Apollo Client is one such tool. It leverages the power of GraphQL API so that it can handle data fetching and management for you. Let's take a look at where Apollo Client will fit within your architecture. Here, we've represented our UI component, which could be framework agnostic, it could be a React app, Angular, Vue, or even your native iOS or Android mobile app, and we also have our Apollo Client represented in the middle and a GraphQL server. Now, the typical workflow is where your UI sends a query to the Apollo Client. The Apollo Client processes this query and requests data from your GraphQL server. Your GraphQL server then responds with the results to the Apollo Client, and here is where your Apollo Client is going to normalize this data, and store the data in an in‑memory cache, and then it's going to send the response to the UI, which is going to update the UI in return. This is going to be the architecture when you integrate Apollo Client within your app. With the Apollo Client, fetching data is done in a predictable, declarative way. You can simply write a query and fetch data without manually tracking loading states. Retrieving data, keeping track of loading states, and error states, and updating the UI are all encapsulated within the useQuery hook. We will learn about this as we progress in the course. The Apollo Client works well with several front end platforms like React, Angular, Vue, as well as native iOS and native Android. It is perfect for modern frameworks, and especially works very well with the React ecosystem. The Apollo Client can perform intelligent caching. It stores the results of your GraphQL queries in a normalized in‑memory cache, and this allows your client to respond to future queries for the same set of data without sending any additional network requests. Isn't that nice? You can leverage the caching feature out of the box without having to write tons of boilerplate code or any extra configuration. It takes the burden off the developer's head and provides them with an intelligent caching mechanism. The Apollo Client is universally compatible. It works with any build setup, any GraphQL server, and pretty much any GraphQL schema. While using Apollo, you can leverage the vibrant ecosystem that it is a part of. Apollo Client comes with the whole ecosystem of tools that integrate with your workflow, providing you a superior developer experience. Apollo Client Devtools, apollo‑codegen, eslint‑plugin‑graphql are some of the commonly used tools. In addition, there are plenty of open source tools as well, and extensions that are built on top of Apollo. Some of the popular ones include AppSync by AWS, apollo‑storybook‑decorator, and so on. The Apollo Client has over 500 contributors and has a large community of developers who are actively engaged in developing tools on top of Apollo and maintaining the Apollo Client in several projects. In our course, we're going to integrate the Apollo Client within a React application, and we're going to then consume an existing GraphQL API. Alright, let's keep going.

Globomantics Demo App

In this clip, I'm going to introduce you to our demo application that we'll be using throughout this course. Welcome to Globomantics Tech Conference. This is our hypothetical tech conference for which we're building a web app. The front end of this web app is coded entirely in React. We have a new GraphQL Server API from the back‑end team that has already been coded and is ready for us to consume. If you're interested in learning more about building the GraphQL API, you can take the course, Building a GraphQL API with Apollo Server by Jonathan Mills. We're going to be consuming the GraphQL API from this course in our application here. Keep in mind that building the server API is not a prerequisite to this course, although, it'll be good to learn more about the GraphQL Server layer as well. Let's look into our requirements for this project. The front‑end React app needs to consume this new GraphQL API for its data needs. APIs for conference sessions, speakers, favorite sessions, and more are available to us. All right, now that you know what the requirements are, let's take a look at our Globomantics Conference App for a quick tour. Here is our fancy conference web app. This is coded in React, and you can see a Conference tab here. This conference page is where you'll see more details about the conference. There are two buttons here, View Speakers and View Sessions. At the moment, they're not functional. As the name suggests, we want to display the speakers and sessions, respectively. This is where our project begins. We're going to consume the GraphQL API provided to us by the server team and build out these pages. The pages are going to display the speakers and the sessions for this conference. Let's look at the features that we want to build for our conference app. We want to display the conference sessions list. The user should be able to click on a session for more information. We should also include a feature to favorite a conference session. Finally, we need to display the speaker information. All right, are you up for the challenge? Let's dive into our installation and setup to get this app running on our machines next.

Download and Setup React App

In this clip, we're going to download and set up our React App and make sure that our Globomantics conference web page is running locally. I'm on VS Code right now, and you can see my project folder, GLOBOMANTICS‑APOLLO. You can find this in the exercise folder, and it'll be named GLOBOMANTICS‑APOLLO before. Notice, we have two folders, api and app. In this course, we'll be working through with the app folder, which is coded in React, and we'll be connecting our Apollo Client within this folder. The api folder contains the GraphQL server code, which we won't be touching. We're going to use that just to install and run our server locally as well. Within the src folder, you can see a pages folder that contains all the pages for this app, and our focus is going to be within the Conference page. Here is where we're going to make queries and mutations from to obtain the conference details. Our app folder contains its own package.json file, and this contains all the packages that we need to run our Globomantics React App. The first step in getting our app running is to install all these packages, so I'm going to go ahead and start installation work. On VS Code, click the Terminal menu, and click on Run Task. This will bring about all the tasks that are available for this project. And we're going to run our installation for the app, which is npm install app. This is going to kick off installation of all the packages that we need to run our React App. Looks like the installation's complete. Now we're ready to start running our React App. So I'm going to go back to our Terminal menu, and Run Task, npm start on the app. You can run these commands from the command line as well. You just need to navigate into the app folder and run npm install and npm start. Alright, looks like our app started, and here we go. We have our Globomantics React App running locally. It's running from localhost:1337 here, and it could be different for you. You can see all our pages that have been built, and the page that we really care about right now is the Conference page. You can see the View Sessions and View Speakers button. You can go ahead and click on them, but it's not going to take you anywhere because we haven't wired up these pages yet. And that completes the installation and set up of our React app. In the next clip, we're going to integrate Apollo client within this React App.

Setup and Run Server

In this clip, let's set up our Apollo GraphQL server and run the server locally. Navigate to the API folder, which contains the Apollo server code. It has its own package.json file with the dependencies that we need, and you can also see there is a schema.js file, which contains our schema for this project. We're not going to be coding anything in the API folder, we're just going to install it and run our server locally. We're going to run npm: install ‑ api. You can also run this from command line by navigating to the API folder and installing the dependencies. Once that's done, we're going to Run Task npm: start ‑ api, and that would start our API locally. Our GraphQL server is now running at port 4000, so lets navigate to localhost 4000/graphql. Very cool, you can see our GraphQL playground here. This is what is really nice about GraphQL, because you have a self‑documenting playground created for you. You can explore the documentation, the schema, all of this from the right side here. You can get an idea of what our documentation looks like, you can look at the different queries and mutations that are available to us, and get a general idea of the schema as well. Notice that the queries and mutations all have their types defined clearly; that's because of the strongly typed schema that we have with GraphQL, which is nice. So this makes it easier on the client to consume the API. So let's play around and write a simple query to see if we get a response back and that will tell us that the API server is working locally. So I'm going to write our first query. I'm going to query the sessions that are available. I'm going to look for the id of the session and the title of the session, and hit play. Well, I see all our results back, and this is a list of all the sessions that we have for the conference. What's nice about this playground is that you can press Ctrl+Space and you can see a list of all the fields that are available for you to query. So I'm going to query one more field, maybe day, and I can see our updated query results back. All right, so now our server's all set up and running locally.

Integrate Apollo Client

We now have our front‑end React App set up and running locally. The next step is to integrate the Apollo Client. To integrate Apollo Client, we need two packages. The first package is the @apollo/client. This contains everything you need to set up the Apollo Client, and we're going to be using version 3.0, which is the latest release for React. The next thing we're going to be using is the GraphQL package. This provides logic for parsing GraphQL queries. So all we have to do is run this command npm install and install both the Apollo Client and the GraphQL packages. Let's go ahead and do that. We are back to our application, and I run the npm install command to install both the Apollo Client and the GraphQL packages. Our installation is complete. If you go back to our package.json file, you can now see the two dependencies have been added. The @apollo/client and the graphql dependencies are seen here. And that's all we need to get started with integrating our Apollo Client now. Now that our dependencies have been installed, the next thing we want to do is to create an instance of the Apollo Client. In this code here, you can see that we've imported ApolloClient and InMemoryCache from the @apollo/client library. And I've created a new instance of the ApolloClient. To this client, we're going to provide the running GraphQL server URL. As you can see here, the URI is passed, and we also created an instance of the cache using the InMemoryCache. Let's go ahead and integrate this within our code. Within the src folder in the app, look for the index.js file. In here, you can notice that we begin our application with the app component, and our React is wrapped around the app component. So I'm going to go into the app file, which is App.jsx, and initialize our Apollo Client here. We're going to import the ApolloClient API, and then we're going to import the ApolloProvider that we'll look at in just a minute. And then I'm going to import the HttpLink. And finally, I'm going to import the InMemoryCache. And all of this comes from the same package, which we just installed, which is the @apollo/client package. We're now going to initialize our Apollo Client here. I'm going to use const, and I'm going to call our client just client. And we initialize it just like you would initialize an object using new. And I'm going to say new ApolloClient. And now I'm going to fill in the parameters. The first parameter that I'm going to pass is to the cache, and this is generally optional. So I want to leverage the InMemoryCache that is available in Apollo Client, and I've initialized that. And the next property I'm going to pass is the link. Here, I'm going to create an instance of the HttpLink. The HttpLink API basically fetches GraphQL results from a GraphQL endpoint over an HTTP connection. And to that, I'm going to pass in the URI. And the URI here refers to the active GraphQL server link that is running. In our case, it is going to be http://localhost:4000/graphql. If you left this field empty, it's going to default it to /graphql. And in addition to that to our Apollo Client, I'm also going to pass another property. I'm going to pass in the credentials. It's basically a string representing the credentials policy that you want for the fetch call. And I'm going to specify this as same origin. The final piece to this is to connect your Apollo Client to the React application, and we do that using the ApolloProvider component. It basically connects your client to React with the ApolloProvider component, and it wraps the React App and places the client on the context. This basically allows access to your Apollo Client from all components. Let's go ahead and do that for our React application. We've already imported the ApolloProvider within our app file. So, I'm going to wrap the ApolloProvider with our AppRouter, which is basically the beginning of our app, and I'm going to pass in the client as well. This is the same client that we just initialized. So I'm going to pass the instance of that client to our ApolloProvider, and now we have our provider wrapping our React application. So what this means is we can access our Apollo Client from any component in this app, and this wraps our integration of the Apollo Client within our React App.

VS Code Extension

In this clip, we're going to add the Apollo extension for VS Code. Before we get started on that, we need to configure our Apollo project by adding the apollo.config.js file. We add this apollo.config.js file at the root level of this project. Many Apollo tools will leverage the apollo.config file, which ultimately reduces the net amount of configuration that you add for your project. In the client project, when we add the apollo.config file, we use the keyword, client. For the VS Code extension to work with the Apollo client, it relies on the knowledge off our GraphQL schema. Here is where we're going to provide our GraphQL server's remote endpoint. In our case, it is localhost:4000/graphql, and that's what we're providing here, and with this remote endpoint, it can pull down the schema from the running server. An alternative to this would be to link a schema from a local file. You can always have a local copy of the schema downloaded somewhere, and you can point to that directory from here. To do that, instead of providing the URL, you would give the property localSchemaFile and give the path to that schema file. In addition to that, in our config file, I'm skipping the SSLValidation. This is basically optional, and it disables the SSLValidation check. Alright, once our apollo.config file is all set up and ready, we're now ready to install our VS Code extension for Apollo GraphQL. On your VS Code editor, search for Apollo GraphQl, and this extension is extremely useful, and make sure you install it. It basically provides an all‑in‑one tooling experience for developing apps with Apollo. It provides you features like syntax highlighting, intelligent autocomplete, performance information, and it also helps you navigate through projects more easily. Let's go ahead and install this extension, and that completes our set up of the Apollo client within our React app, and we also have the config file set up, and the VS Code extension installed. At this point, all we have to do is begin coding.

Summary

And that brings us to the end of our first module. In this module, we learned about what the Apollo Client is and why we're going to be using the Apollo client to consume our GraphQL Server. We then toured our demo application, the Globomantics Conference App. Then we worked on setting up our React application and the GraphQL Server. We then worked on integrating the Apollo Client and set it up for our React application. Finally, we installed our VS Code extension for the Apollo Client. At this point, we're now ready to start consuming the GraphQL API from our client. Let's get going on to the next module.

Exploring the GraphQL Schema with Introspection

Module Overview

Exploring the GraphQL Schema with Introspection. In this module, we'll learn about what is introspection in GraphQL, and why it is beneficial. We'll learn about the introspection in GraphQL system to retrieve a GraphQL schema's type. We'll learn what queries and mutations an API supports. We'll also query property types. We'll then learn about retrieving deprecated fields from the schema. We'll learn about query directives. And finally, we'll use introspection with the Apollo client.

Introspection

What is introspection in GraphQL? The strong type system in GraphQL gives the ability to query and understand the GraphQL API schema in detail. GraphQL provides a way for clients to discover the resources that are available in the GraphQL API schema via what's called an introspection system. The introspection system is a feather in the hat for GraphQL, it's very useful, and we'll see why. Imagine you're building a client application and have to consume a brand new API to build the front end. We typically have to go over plenty of documentation to understand the API. We may have to even get into meetings with the back end team to further understand the resources that are available to us. In GraphQL, introspection can provide clients a view of the available resources in the schema. This is very beneficial for the clients consuming the API. Clients or front end teams can get a complete understanding of what is available to them in the schema and can plan accordingly. Clients don't have to go over extensive documentation to find out details about the schema, they can instead introspect the schema via simple queries. The introspection feature in GraphQL allows us to build awesome tools, tools like GraphiQL and GraphQL Playground leverage the power of introspection. These GraphQL tools use the introspection system to provide features like self documentation, autocompletion, code generation, and other exciting features. All right, let's get going and introspect our Globomantics schema to learn more about what resources are available to us as a client. We're going to write our introspection queries against the server schema, which is running on localhost:4100, and as we discussed, what's cool is we're going to use the GraphQL Playground, which itself is a tool that was built because of GraphQL's introspection system. Let's get started.

Retrieve a Schema's Type

Retrieve a schema's type. In this clip, we're going to use introspection to retrieve the Globomantics schema's types. Make sure your server is running, and navigate to http://localhost:4000/graphql. You should see the GraphQL Playground running like this. The first introspection query that we're going to write is to retrieve the schema's types. This is a good starting point to know what types are available in the schema. To do this, we can simply write a query using the double underscore schema field. This should be available in all GraphQL APIs. I'm going to define our query and give it a name retrieveAllSchemaTypes. And I'm going to query for the field \_\_schema, and this will access the current type schema. And within the schema, I'm going to look for types. This is going to list all the types supported by this schema. And I'm going to look for the name of these types. And we're done with our query, so I'm going to hit play. Here we are. We get the response back from our server, and you can see that we have plenty of types in the schema. Let's go over them. Types like ID, Int, String, Boolean, are all built‑in scalar types in GraphQL. We also see some custom types here that have been built in for the Globomantics project like Session, Speaker, User, SessionInput, and so on. Let's scroll down a little bit more. The types that you see here that start with two underscores like \_\_Schema, \_\_Type, \_\_TypeKind, represent the introspection system. Very cool. So we know all the types supported by the schema, let's try to learn more about them. We can always query for more fields than just the name, so I'm going to go ahead and add to our query the field kind. That could give us more information about our types. As you can see here, the kind tells us that some of these types are SCALAR, some of the types are INPUT\_OBJECT, or just plain OBJECTs. You can also query for the description, and we can see if there's a description available with these types, so we can get more information about that. And all the scalar types, by default, have those descriptions. And if the schema did have a description for the user‑defined types, that would be something we would see here as well. All right, so now we know how to retrieve a schema's type using introspection.

Retrieve Supported Queries and Mutations

Retrieve Supported Queries and Mutations. Now that we know the type supported by the schema, we're at a point where we're interested to learn about the queries, as well as the mutations that are available to us in the schema. We use queries to query for data from the server, and we use mutations to add or update the data. To do this, we're going to use another introspection query. I'm going to call our query retrieveQueryAndMutationTypes. I'm going to use the \_\_schema that we used from the previous example, and within that, I'm going to look for queryType, and this is going to tell us all the different types of queries supported by this schema, and I'm going to query for fields, and within fields we're going to look for the name of the queryType. It might be useful to also add the description here, and we're done with our query, and we can hit play now. You can see our response that has come back, and you can see all the different types of queries that are available as a part of the schema, we have sessions, sessionById, speakers, speakerById. Notice that we don't have a description string along with any of these queries. What this means that originally when the schema was written, a description was not added to these queries. The next thing I'm interested from the client perspective is to know the mutations that are available to us from the schema. Within the same query, I'm going to query for the mutationType, and within that, I'm going to query for fields, and the field that we're looking for is name and description. We've got our JSON response back. You can notice that we have results for both the queryType, as well as a mutationType. The mutations that we see are createSession, toggleFavoriteSession, signUp, signIn, and so on. What's interesting with GraphQL is even if you're looking for multiple fields, you can combine them all within one query, just like we did right now with our introspection query, and we'll look for both queryType, as well as mutationTypes. Note that because of tools like GraphiQL Playground that we're using right now, you don't have to really write these introspection queries to learn about what is available in the schema, you can simply open the documentation and explore. You'll see the list of queries and mutations available in the schema right here within the documentation tab, but remember, that the reason these are available to us so easily is because these tools were built using the introspection system in GraphQL. This is why it's a good idea to learn about the introspection system and understand the big picture of how all of this comes together. All right, let's go ahead and write some more introspection queries.

Query Property Types

Query property types. We now have the list of types supported by the schema, and we're also familiar with the queries and mutations that are available to us. Now I'm interested to know more about an individual type. To do this, we can use the \_\_type from the introspection system. I'm going to name my query retrieveTypeInfo, and in this query, I'm going to query for the field \_\_type. That's going to give us more information about a specific type. We passed to this the name of the type that we're interested in as a parameter. In our case, I'm going to query for the type Session. Within this, we're going to query for the fields, and all I want is the name of the fields for the session type. All right, our query is ready now, and let's hit play. Our JSON response is back, and you can see the list of fields associated with the session type, like id, title, description, room, day, format, and the speaker. So this makes sense to me because a conference session probably needs to have all of these data items. I've also queried for the description, and at this moment there is no description associated with any of these fields. Let's make this more fun and query for one more type that we're interested in, which is the speakers. And again, all I want to know is the name of the fields that are associated with the speaker type. But, hey, I noticed that there seems to be a syntax error here. It's pointing out to us that we're using the same field \_\_type with different parameters, and that would be a conflict, so I'm going to provide an alias to it. Aliases let you rename the result of a field to anything you want, and in our case, we rename them to a sessionInfo and speakerInfo. And our JSON response has come back without any errors. I did notice that our speakerInfo has come back as null. Maybe our type was just Speaker and not Speakers in a plural way. I've corrected that and you can see we've got JSON response back, and we have results for both the speakerInfo and the sessionInfo. In GraphQL, you can directly query for the same field with different arguments, which is why we had to overcome that by giving aliases here. All right, so now we have information about individual types. There are some times when clients may be interested in the deprecated fields. In our case, we're building a brand‑new application and we don't care about the deprecated fields. But in case we're a client that was already consuming this GraphQL service schema, there are times when the server could deprecate some fields, and as a client, we may want to know about it. Let's extend our previous introspection query and look for the deprecated fields as well. To do that, I'm going to pass the parameter includeDeprecated to our fields, and I'm going to give the Boolean value true. This means that the results are also going to include deprecated fields. And in addition, I'm also going to query for the value isDeprecated and the field deprecationReason. So if our query results are going to have deprecated fields, then the isDeprecated is going to show us a Boolean value of true, and it may also have a deprecation reason. So let's look at our JSON response to see if there is any deprecated field. There is one I see right there, which is the field track, and it has returned the Boolean value true for isDeprecated, and it also has a deprecation reason that says there were too many sessions that do not fit into a single track, and that makes sense. All right, we can extend the same thing for the type Speaker as well. So when we go inside our speakerInfo, we can see if there are any deprecated fields, and it doesn't look like there were any deprecated fields. With this, we've learned about how to introspect our schema for specific types.

Query Directives

Query directives. In this clip, we're going to learn about how to use the introspection system to query for directives in the schema. Directives provide clients the flexibility to modify the results of queries based on the criteria that the clients provide. Let's get started with our query here. I'm going to call our query retrieveDirectives. We're going to query for the \_\_schema field, and within that, I'm going to look for directives, and this is where a list of all the directives supported by our service schema can be queried for. And within the directives, all I want is the name of the directive. So our schema supports four directives that you see from the JSON response, this cacheControl, skip, include, and deprecated. Now, if we want a little bit more information about them, we can query for the description to see if they have a description tied to it, and I do see a description for most of them, so let's go over them. The skip directive is used on fields of fragments, and it basically allows clients to exclude fields based on a condition. Similarly, the include directive does the opposite of skip; it allows clients to include fields based on a condition, and the deprecated, as the name suggests, is for deprecating an element, so when there is an element of a GraphQL schema that is no longer supported, we use the deprecated directive. So what is cacheControl? We don't see a description for it, but in production apps, we often rely on caching for scalability, and the Apollo server provides a cacheControl directive to handle cache control parameters on individual GraphQL types and fields. Alright, so now we can extend our query to get more information. I'm going to query for the field args, which means arguments, and I'm going to look for the name of the argument and the description of the argument. So let's hit Play to see what our JSON response looks like. In our JSON response, you can notice that each of our directives actually has taken an argument as well. The cacheControl takes in an argument of max, age, and scope. Skip and include take in an argument of if; it basically means that you can pass a Boolean value to the if, and your directive can be either skipped or included when it is true, and the deprecated directive takes in an argument reason, and this is where you can explain why a certain element was deprecated. Now, as a client, we're familiar with all the directives that are supported by the schema. We have skip, include, deprecated, and cacheControl, but let me show you how we can use a directive within a query. So I'm going to write a simple query to retrieve sessions here, and you can see that we've retrieved the session's title and the day. Now, let me show an example of how to include a directive. Within the query variable, let's declare a variable called with\_day, and give it a value, true. Now in our retrieveSessions query, let's pass an argument to it and that would be the with\_day variable, and I am going to also define that as a Boolean value. I want our retrieveSessions query to include the field day whenever the client wants to, and the client can use this with\_day boolean variable to do so. To use a directive, we use the @ symbol, and I'm going to use the include directive for the day. And we also have to pass the argument for if, as we saw in the introspection query, and I'm going to pass to it the value true or false. And over here, we're going to have a variable with\_day that we're going to pass. Now let's hit Play and see our response. Our JSON response is going to include both title and the day. Now if we want to change our value of with\_day to false and hit Play again, there you go. You see that the day is not included anymore, and this is how directives provide flexibility. Alright, so now you have a good general idea of what directives are and how to obtain information about the directives.

Apollo Codegen to Introspect Schema

In this clip, we're going to learn about using Apollo Codegen to introspect a schema. So far in this module, we learned about GraphQL's introspection system and its benefits. We also wrote some introspection queries to better understand it. Using Apollo client, there is a simple way to create a JSON introspection dump file for a given GraphQL schema. This will contain all the possible information that you need through introspection. To do this, first install the Apollo Codegen tool. You can run this command npm install apollo‑codegen at the app level. Once the installation is complete, run this command to generate the JSON introspection schema dump file. The input schema can be fetched from a remote GraphQL server or from a local file. In our case, we can give the GraphQL server location, which is localhost:4000/graphql. Here I've given the name schema.json for the output file. This is up to you, and you can give any name for this file. I'm back to my VS Code here, and after running these commands, if I open my package.json, I should see the Apollo Codegen library added as a dependency here. We see it right here. I also see the new schema.json file, which has been generated, and this contains the complete introspection dump, which you can see here. You'll be surprised to see how big this file is. Well, there's a lot going on here, and as you can see, we have some of the things we already talked about, like the types, you can see sessions, speakers. And it has all information, like the kind, the type, the description, and it's basically all the possible queries that you can think of within the introspection system. So at this point, we have everything we need to know about this schema, and we know about it in detail after looking at this schema.json file. All that's left to do right now is to start writing queries against our server and consuming the data to build our client Globomantics application.

Queries - Retrieve Data from GraphQL

Module Overview

In this module, we're going to be learning all about GraphQL queries. Queries in GraphQL is used to retrieve data from the GraphQL server. We're going to learn about GraphQL queries using the Apollo client for our React app. The topics that we're going to be learning in this module are introduction to the useQuery Hook that we can use in our React application, writing queries with arguments, writing queries with variables, error handling in queries, reusing fields with fragments, aliases in GraphQL, and dynamic queries with directives. We're going to be demoing all these concepts using the Globomantics conference site, and we're going to be building the site together as we retrieve data from the GraphQL server. By the end of this module, you'll be proficient to write GraphQL queries for your React application using Apollo client. Let's get started.

Queries

We're about ready to start writing our first GraphQL query. A recap on GraphQL queries. GraphQL queries provides clients the power to ask for exactly what they need and nothing more. This is what makes GraphQL powerful. You can combine all the data that you need, even in one single request, thereby reducing multiple round trips. Queries are essentially tailor made per client. With the Apollo React integration, we're going to use what's called the useQuery Hook. This is a React hook, which is the API for executing queries in an Apollo React application. We're now going to create a sessions page in our Globomantics app. This page will be within the conference tab and will contain the session details. For now, we'll stop this page with hard coded data and make sure the page is up and running. Once it's ready and running, we'll work on retrieving the data from the server and display the sessions for the conference. We're back in our Visual Studio IDE, and what we're going to do right now is create a sessions page. So go into the app and within the src folder, go into pages, and here within the conference, I'm going to create a new page for sessions. I'm going to copy/paste a code snippet here, and all I have here is basically a simple React, UI. It's going to create a sessions page and it's a functional component right here, and you can see that we have a session item that we're displaying. And as of now, our session item is hard coded, we don't have any data that we've retrieved from the GraphQL server yet, and we've just hard coded a title, Day, Room Number, and Level. And now, I'm going to include this in our route along with the conference pages, so go to the conference page, I'm going to import our sessions page here. The sessions page that we just created is imported, and I'm going to include that as a route here. We have to define the path to our route, in our case it's going to be /sessions, and within this route, we're going to call the sessions component. Fair enough. Our code is running at localhost:1337, so let's go check out what we have so far. Here is our Globomantics app and we have this conference page, we just included our sessions within this conference page, so clicking on the View Sessions should take us to the sessions page. And as we know, we don't have any data that we've retrieved from the GraphQL server yet, this is just a placeholder with some UI that I already created. Let's now get started with actually retrieving the sessions data and displaying it on this page.

Query to Retrieve Sessions

In this clip, we're going to write a query to retrieve sessions from the GraphQL API. In this demo, we're going to write our first simple query using the Apollo Client. And we're going to query for the sessions data from the Globomantics API. We're going to query for this data using the useQuery hook. Once we retrieve the sessions information, we're going to display this data on the Globomantics website. I hope you're all set to start writing our code to retrieve the sessions data. First things first, we're going to import a bunch of things from the Apollo Client. We're first going to import the gql, which stands for GraphQL, and also the useQuery hook. And these are all coming from our Apollo Client package. We're now ready to first define our query. As we talked about, we're going to query the sessions data, so I'm going to create a const and give it a name, SESSIONS. And we're going to use the const GQL, which comes from the Apollo Client library. And within here, we're going to type out our query. So we're going to give a name to our query, I'm going to call it sessions. We're going to begin by querying for the field sessions. And as you can see here, our VS Code is prompting to us using the autocomplete here. And this is because we have the Apollo plugin installed, as well as the Apollo config, which has information about the schema. So any field you want to query, you can see that we have the autocomplete feature, which is nice. So we're querying for id, title, day, room, and level. And these are all information pertaining to the sessions. Our initial query is complete, and this seems like quite a bit of information about the sessions that we can display on our page. The next thing we want to do is to execute this query and obtain the JSON response. I'm going to do that within a function, and I'm going to call it SessionList. And within this function, I'm going to execute our query that we just defined and store the JSON response. This is where the useQuery that comes with Apollo Client version 3 for React comes in handy. We're going to define a const loading and data, and we're going to store the response from the useQuery within these constants. And the useQuery hook is going to take in the query as the input and our query name here is SESSIONS. So what this essentially does is it's going to execute our SESSIONS query, and the JSON response is going to be stored within the data. We also have a const loading, and that's going to be a Boolean value, which will be true while the GraphQL query is still loading. So that would help our UI where we can either show some kind of spinning wheel or just the message that the query is loading. For now, I'm going to keep it simple and say Loading Sessions if loading is true. And once it's done loading, we're going to go and iterate through our data. We're going to go into the data, and within the data, we're going to get a JSON response, which would be sessions, and I'm going to use the map function to go over each item in the session. For each session, we're going to call the component SessionItem, and we're going to pass a key to it, which would be the session.id, and the rest of the session information will be passed on as well. Now, I'm also going to go ahead and make a small change to our function Sessions because earlier we were calling the SessionItem in here, so I'm going to change that to SessionList, which then calls the SessionItem per session. Now we need to replace all our hard‑coded dummy data to the real data that we got back from the server. So the SessionItem is going to take in the session data, and I'm going to extract all the information that we have within session, the ID, the title, day, room, and level. Now let's go ahead and replace all of those hard‑coded data with the constants that we just extracted from the Sessions object. I'm replacing the title and then now I'm adding the day for the session. We're then going to go ahead and add the room number for the session, and we'll also include the level for the session. All right. I think we're all set now. A quick recap here. The first thing we did was to define our query. We wrote a query for the sessions. The next thing we did was we used the useQuery hook to execute our query and store the JSON response. And this JSON response was passed to the SessionItem component, and this component extracted the id, title, day, room, and level of each session and it's going to display it. All right, let's go ahead and see how it looks. We're back in our Globomantics app. And remember our sessions page is within the Conference. I'm going to click on View Sessions, and there you go, you see a list of real data that just came back from our server, and each title here represents a session, which contains a title, day, room, and level. And well, here we are, we wrote our first query and retrieved real data from our GraphQL Server. Let's keep going and learn more about GraphQL queries.

Query with Variables

In this clip, we're going to learn about querying with variables. All right, our conference app is looking good now, and we've retrieved the sessions for the conference from the GraphQL server. Hey, but this list looks huge. I think it would be really useful for the conference attendees to filter by each day. It's a three‑day conference, and we have sessions on Wednesday, Thursday, and Friday. I've added three simple buttons here on our sessions page. Pardon me if it doesn't look very pretty. I want our GraphQL query to be modified such that it only queries for the day that the user selects. So the idea is clicking on Wednesday should only display the sessions on Wednesday, and clicking on Thursday should only display sessions on Thursday. Cool. How do we solve this? We can solve this problem by using variables in GraphQL queries. Arguments to fields in GraphQL can be dynamic. Just like we saw here, we don't want a hardcode the day as Wednesday, Thursday, or Friday. It's dynamic based on the user's input. GraphQL uses variables to query for dynamic values. In this demo, we're going to extend our query to include variables. We're going to demonstrate including variables for the day of the conference. All right, we're back to our Sessions page here. You can notice I have made some modifications. I've added the three buttons here for Wednesday, Thursday, and Friday, and I'm also using State, the hook useState, to set the day that the user selected. And this day that we selected is then passed to the SessionList. All right, all of the react UI work is done, but now I need to update our query to take in the day as an argument and also execute the query based on the dynamic variable. Let's look at how we can do that. I'm going to go ahead and update our sessions query. Currently, our sessions query does not take in any argument, so we're going to pass an argument to our Sessions query. And remember, this argument is going to be dynamic. We're going to make that possible using variables. Variables are declared in GraphQL using the dollar sign, and I'm going to call our variable day, and I'm also going to make sure that a type is given to it, which is string. And what this means with the exclamation mark is that we always need to pass the day variable to the sessions query. All right, the next thing we're going to do is to pass the day to the SessionList component. I've already sent it to the SessionList component, so I'm making sure that this function argument has day in it. The next thing we're going to do is to update our useQuery hook. Our useQuery hook currently just takes in the name of the query. We're going to accommodate a variable to it, so we're going to use the keyword variables, and to that, we're going to pass down the day that we have within this function. And that's it. I think we got all our pieces done. We updated our query, and we also updated our useQuery hook to pass in the variables, and our UI is already complete. So at this point, I think we're all wired up, and it should work, so let's go ahead and check out our demo. The Globomantics app is loading, and I'm clicking on View Sessions. Right now we don't see any sessions until we click the button, and after clicking Wednesday, you should just see sessions for Wednesday. Yep, that works. And I'm going to click on Thursday, and I see only Thursday sessions here, so seems like our query worked out, and Friday seems to work too, great. Now, if you notice, eventually when I click on the same buttons again, it doesn't really take any time to load. The first time when this website loaded and I clicked on the buttons, it took a fraction of second to load these queries. Well, why do you think that's the case? That's because the Apollo cache is at work, and our data got cached without any additional work from our end in terms of code. Isn't that neat? So all of your loads now are instant because it loaded it the first time around and cached it. Great. So in this clip, we saw how we can pass variables to our queries, as well as passing variables as a configuration option to the useQuery hook. All right, let's learn more in the next clip.

Default Variables

In the previous clip, we learned about including variables to our GraphQL queries. In this clip, we're going to learn about adding default values to our variables. The very first time we open our Globomantics conference app and click on View Sessions, you'll notice that we don't have any sessions displayed. This is because we just implemented the filter function, where we only display sessions based on the day that we selected. Since no day is selected here, we don't display any sessions, but is that really intuitive to the user who's opening our conference app for the first time? Probably not. So, we probably need to have some default variable assigned, meaning that we need to have a default day assigned so that when they open up the conference app for the first time, they still do see some sessions, and then depending on the button they click, we're going to display the session for that specific day. To do that, we're going back to our sessions.jsx file, and I'm just going to make a quick tweak here. I'm going to check the day that has bean passed to us in the SessionList function. I'm going to see if it is an empty string, and if it is, then I'm going to give it a default value, and I'm going to pick Wednesday, which will be the first day of the conference. This is going to default our day to Wednesday if no day was passed by the user. So let's go back and check it out, and back in our Globomantics app and clicking on View Sessions does bring back the sessions for the very first time, and it's going to show us the Wednesday sessions, which is what it's defaulted to. And, the user can always switch around and see the sessions for the other days.

Error Handling

In this clip, we're going to learn about handling errors, my favorite topic. Errors are there everywhere. Think of all the times you've been browsing through the internet and run into a dead screen, you don't know what's going on, and it's just errored out, right? Errors can happen when we're executing GraphQL queries for several reasons. Maybe the query was incorrect, maybe the server isn't responding, or there could be some simple network issues. Whatever the situation may be, we as developers need to handle these errors gracefully so that our end users are not left frustrated. The Apollo useQuery that we use to execute our queries returns a result object with error and loading properties. We already looked at the loading property in the previous clips, and now let's go ahead and utilize the error property. We are back to our sessions.jsx file, and right here where we execute our use query. As you can see here, we extracted the loading and the data property. I'm going to include the error as a property here as well. And, there are fancier ways to handle the error, but what I'm going to do right now is just check if there is an error and display an error message, for demonstration purposes. And this simple message will do for now. Let's think of a way to error out our GraphQL query. So I'm just going to type an incorrect query so that we see an error on our Globomantics site. Maybe I'm going to just query for a field that doesn't exist and see what happens. So we're back to our Globomantics site, and I'm going to View Sessions, and there it is, it says there's an error loading the sessions, and no matter what, it's going to display that message, and you can assume that this message can be utilized for any kind of error. It could be a server error or a network error, it doesn't matter, but we just need to gracefully error out instead of keeping the user thinking about what's going on. If you're curious to see what this looks like on the console, you can always right‑click and inspect. Make sure to go to the Network tab, and this is where you're going to see the GraphQL server requests and responses, so let me hit on a certain tab, there you go, you see the red here indicates an error. I can click on it to see more information, and we see there is a Status Code: 400, and that's an error response from the server, and it says it's a Bad Request. We can also scroll down to see if there's more information here. Here you can see the Request Payload, you can see what our operationName was, you can also check out the query that we passed in, and as you can see, the variable is also passed in here. Simple and doable, isn't it?

Query to Retrieve Speakers

In this clip, let's write a query to retrieve the speaker information for our conference app. First things first, we need to create a Speakers page. To do that, within our conference folder, create a new file called speakers.jsx. I'm going to paste a code snippet here, which is going to be the UI for this page. It doesn't have any GraphQL queries written yet, and just like we did for the Sessions page initially, I'm just hardcoding the speaker information right now, and set up the UI component for the Speakers page. Now what we're going to do is define our speaker query. We'll have to import our GraphQL function and the useQuery hook from Apollo Client. Once we have them imported, I'm going to define our speaker query and give it a name, SPEAKERS, because we're going to retrieve all the speakers for this conference. And remember to pass in your GraphQL query within the gql function that comes with Apollo Client. And in here I'm going to give our query a name called SPEAKERS. And we can start querying for some speakers fields, so the first field were going to query for is the SPEAKERS field. And don't worry if you don't know the schema by heart. You can obviously rely on the auto‑complete. I'm going to look for the id, name, and the bio fields for the speaker. And we also want to know the sessions that the speaker is going to present at the conference, and the speaker could be presenting multiple sessions, so we're going to look for sessions, which is an array, and within the sessions, I'm going to query for the field id of the session and the title of the session. Perfect. So our query is all set up. The next step we're going to do is to execute our query, and we'll be using the useQuery hook. The useQuery hook does return some objects, like loading, error, and data, so I'm going to extract all of that information. And to the useQuery hook, we're going to pass in our query, which is the SPEAKERS query. And we're going to show the loading state and the error state on the Speakers page as well. (Typing) All right, so the next step is to iterate through our JSON response, which is stored in data and replace all the hardcoded data with the actual data that we get back from our GraphQL server, and to do that, I'm going to iterate through the data object. And within the data object, we're going to iterate through the speakers, and I'm going to use a map function here, and I'm going to iterate and display each speaker information. And within each speaker, I want their id, name, bio, as well as the sessions that they're going to present. All right, and now we're going to do the same things as we did in the Sessions page. We're going to replace all of this hardcoded data, so we're plugging in the name off the speaker, and next, we're going to add their bio information. And with regards to sessions, as we talked about, one speaker could be presenting at multiple sessions, so this is an array of sessions, so we have to iterate through the sessions per speaker. So I'm going to do that and iterate through all of the sessions that one speaker is presenting. So I'm going to use the map function again and iterate through each session, and within each session, we're going to display their title, and that would be sufficient for us right now. Perfect. So our query for the speakers is complete, and we also have our useQuery hook that'll execute the query, and we parse through the data and display it in our UI. So our Speakers page right now is complete. Now I need to hook in the Speakers page within our conference app and add a route for it, so let's go ahead and do that. Within our conference page let's go ahead and import our Speakers component. Once it's imported, we need to include a route for the speakers and provide a path for it, so I'm going to include the Speakers path as well as include the Speakers component here from the conference page. We've now retrieved the speaker information from our GraphQL schema, and we're displaying it here on our Speakers page. Each tile here, just like the Sessions page, represents a speaker. We're showing their name, their bio, and all of the sessions that each speaker is participating in. So far, we've written two queries against our GraphQL Server, one to retrieve sessions and the other to retrieve the speaker information. Let's see what other fun things we can do with GraphQL queries.

Fragments

Fragments in GraphQL. Fragments in GraphQL are reusable units in GraphQL. You can think of this similar to a function in other programming languages. What do we do with functions? We take logic that can be reused and place them within a function, and then we use this function across several places in the code base. The same logic applies to fragments in GraphQL queries. We can build sets of fields and reuse them across multiple queries using what's called fragments. Fragments make your GraphQL queries more readable, and if you were to change something within a fragment, then it applies to all the other queries that use the fragment. In this demo, we're going to learn about using fragments within our GraphQL queries, and to do that, I'm going to use fragments in the speakers and speakerById queries. Exciting, let's go ahead and start learning that. So far in our Globomantics conference page, we've hooked up queries to View Sessions and another query to View Speakers, but I want to improve the usability here for our audience, where when they open a session, they can also look at who the speaker is and look at the speaker's profile. So I want to tie the session to a speaker. So on each session, I want to display the name of the speaker, and when they click on the speaker, they can get more information about that specific speaker. To do that, we need to write another query, which is queryBySpeakerId. In our speakers page, let's go ahead and write our second query for the speaker, where we retrieve a specific speaker given their id. We're going to pass the query within the GraphQL function. Let's give our query a name, speakerById, and we're going to query for the field speakerById. We need to pass in the id of the specific speaker to this query, so let's go ahead and pass that as a variable to the speakerById query, and within the query, I'm going to query for fields id, bio, name of the speaker, as well as the sessions that the speaker is going to be presenting. Perfect. Our query is now set, and I already plugged in the React UI for this with dummy data. So the next step that we're going to do here is extract the speaker ID. Now, for this I'm going to use the useParams hook that comes with the React router, and what this essentially does is whenever the user is going to click on a specific speaker, we're going to pass that speaker's id, we're going to grab that and pass that through the URL, and that's what over here we're going to get within the SpeakerDetails component, it's going to obtain that speaker id. You're familiar with the next step, we're going to execute our query using the useQuery hook, we're going to obtain the loading error and the data objects, and to the useQuery hook we're going to pass our query speakerById, and to this we're also going to pass the variable id, and here, the ID is going to be the speaker id that we obtained from the useParams hook. Perfect. I'm going to copy/paste our loading and error states. When it's loading, we're going to show a loading message, and when it errors out, we'll be showing the error message, and once that's done, at this point, all we need to do is extract our speaker object from the data response that we got and pass that down to the UI. Now, let's take a look at both our queries here. Both the speakers and the speakerById query seem to be using the same fields, they're querying for id, bio, name, and the sessions, and this is where maybe we could use fragments. Fragments come into play here where we could put all of these reusable fields within a fragment and reuse that fragment across these queries. So let's get started and learn how to write a fragment using Apollo client. The first step is I'm going to define a constant here for our fragment, and I'm going to call it SPEAKER\_ATTRIBUTES, and fragments are also supposed to go within the gql, or the GraphQL function, so I'm going to go ahead and define our GraphQL function here. You will then use the keyword fragment, and give a name for your fragment. So in our case, I'm going to call it SpeakerInfo, and that would be the name for our fragment, and you also need to define on what object you're declaring this fragment. So, we're going to declare it on the Speaker object, and luckily VS Code does help us out here, so we did get that autocomplete. And now, it's simple, you just copy all the fields that you want that needs to go into the fragment, and paste it within your fragment. So I want the id, name, bio, sessions, and the sessions id and title. So, our fragment is defined, and this is basically all the reusable fields from both those queries. Now, all you need to do is call this fragment within our queries. To do that, use the ... operator and provide the name of our fragment, which is SpeakerInfo, and also within the GraphQL function here, you need to define from where this fragment came from, so our fragment came from the SPEAKER\_ATTRIBUTES, and that needs to be provided here. Repeat the same steps for our next query, just use the ... operator and invoke a fragment, SpeakerInfo, and provide context for that, so provide the name SpeakerInfo so it knows which fragment we're talking about. Perfect. So, this is an example of a co‑located fragment where the fragment is located within the same file as our speakers component. All right, so you can look at how concise our queries look now, it's all simple and neat, and if you ever wanted to have more fields, you will go back into your fragments and include those fields, so both those queries can take those. All right. Our next step is to include the speaker component within our route. So in the conference page, I've included the path for the speaker, along with the speaker id so the URL is basically going to be the path/speaker/ that specific speaker's id, which is what we were going to be passing through the useParams into our speaker component. We talked about having this from the sessions page where I display each session along with the speaker name, and clicking on that speaker name gets us to the speaker information. So to do that, I've updated our sessions query to include the speakers id and name, so this is going to have our sessions query along with the name of the speaker as well, which makes sense. And, I've also updated our UI, where I've added a link, and this link here, as you can see, is linking to the speaker page. So it's going to obtain the id of the speaker, and when we click on that, it's going to link to the conference speaker page. All right, let's check out if all of this worked out. We're back to our conference app, and within our sessions page now, you can see we have each session along with the name of the speaker as well. And, let's try to click on one of these speakers, there you go. Clicking on the speaker page took us to the specific speaker's details. It basically took us to the URL speakers/ the speaker id that displayed that specific speaker's information, along with the session that they were presenting. Perfect, and we've written our queries in a reusable manner using the concept of fragments. Let's go ahead and learn more about queries in our next clip.

Aliases

Let's learn about a neat trick and GraphQL using aliases. So far, we've learned about arguments in GraphQL queries. We can pass arguments to fields to filter out the resulting data. In the previous clips, we passed arguments to filter out data based on the day of the session. We also passed arguments to our query where we filtered out speakers based on their ID. Every field and nested object in GraphQL can get its own set of arguments, but we can directly query for the same field with different sets of arguments. This will result in a conflict and a syntax error in the GraphQL query. To overcome this, we can use aliases. Aliases let us rename the result of a field to anything we want. By simply giving a different name, we can get rid of the conflict in the GraphQL query. In this demonstration, we're going to use aliases in our GraphQL queries. We're going to look into sorting the sessions based on the level of the session, and use aliases within the query. Let's jump right in. Let's get into our Sessions page for the conference. So far, we've built a pretty sleek website with our sessions displaying, and we can filter based on the day of the session as well. Another additional feature that I'd like to have here is that all the levels here are mixed up. If you notice, we have some intermediate talks, we have some introductory talks. There are also some advanced talks in the mix, and back again to some introductory talks. I'd like to sort this page to display introductory talks first, followed by the intermediate talks, and finally, the advanced talks. That way, as a user, when I'm scrolling through the sessions, if I care only about the advanced talks, I'm going to go all the way to the bottom, and if I care only about the introductory talks, I'm going to go all the way to the top, and they're sorted nicely, instead of being mixed up. Alright, so let's go ahead and get this coded. We're back in our sessions.jsx file, and I'm going to go ahead and update our sessions query. We're going to pass in another argument to our query, and this time, it's going to be the level of the session. So I'm going to look for just the intro‑level sessions. By doing this, we're filtering out all the other sessions, and it would only display the intro‑level sessions. Alright, so I'm going to copy this and try to do this again, but this time, I'm going to pass another level. I'm going to pass the intermediate level as the argument to the Sessions field. As soon as you do that, you're going to encounter an error. It tells us that the fields session conflicts because they have differing arguments. Just like we talked about, we have to use aliases when we have the same field, which takes in different arguments. Alias is a simple concept. I just have to give this a different name. I'm going to rename the field to intro, and I'm going to rename the second field to intermediate, and I'm going to use the colon right there. By doing that, you've defined an alias and got rid of the conflict that GraphQL just showed us. I'm going to repeat the same steps one more time, and this time, I'm going to call our alias as advanced, and I'm going to pass to it the level Advanced. Alright, so now our GraphQL query is ready. We have queried for the sessions, and we've split them up now into three different aliases: intro, intermediate, and advanced, and each one of them is going to contain the three different levels of sessions. And you're going to look at this and be like, hey, there is a lot of repetition. When that happens, we're going to use a fragment to consolidate all of those reusable fields. So I'm going to create a fragment, just like we did for our speakers. We're going to create a fragment, and I'm going to call this session info, and basically copy all the fields that were part of the session's query and put it within the fragment. Now we can get rid of all these fields from the query and instead, call the fragment using the ... operator, and call our fragment session in full. And we can do this for all the three sections that have the repeated fields. By doing this, you'll start to realize how concise our query looks now. One last thing we need to do is add the reference to where the fragment was defined, which is SESSIONS\_ATTRIBUTES. Take a look at our query now. It was one huge query, which has now been completely concised, and all of the reusable fields have been wrapped into this fragment, which is why I really like fragments, because it makes our queries look beautiful. So, yes, we have our aliases defined. We have the intro, intermediate, and advanced. And now we need to utilize them, so I'm going to go back to where we iterate through the sessions. Now, we're no longer going to get data.sessions. Instead, we're going to have data.intro and intermediate and advanced. So we're going to replace our return statement right here. I'm going to get rid of this return statement. We have a results array now, and I'm pushing the results to this array. First, we would have the intro results, followed by the intermediate, and then the advanced. By doing this now, our site is going to be sorted out, so the levels are going to be displayed one after the other in a sorted manner. Obviously, if you want to improve the UI, you can then have a button to just display intro levels or intermediate levels. Alright, so I also think that our query needs to include the time of the session. I just updated that to have the time, and we can include that in our UI as well, so it'll give us a little bit more information about the conference session. Perfect. We're back in our Sessions page now, and lo and behold, let's see if our levels have been sorted out. I only see introductory and overview sessions so far. Well, I see the intermediate sessions right there. Perfect. So first, intro followed by the intermediate sessions, and right at the bottom, I see the two advanced levels. Alright, so it worked out well. We sorted out our data based on the level of the session, and we were able to achieve that using arguments and aliases to hold onto multiple arguments for the same field.

Query Directives

In this clip, we're going to learn about a topic that may come in handy to you while executing GraphQL queries. It's called querying with directives. Directives can help you execute dynamic queries. We so far saw how we can pass arguments to the query and also pass variables as arguments. The next step in this is also executing queries that could be dynamic in nature, meaning that the result off your JSON response sometimes may include certain fields and sometimes may exclude those fields. You can achieve this by using what's called the directives. We're back to our Sessions page here, and I'm going to extend our sessions query and include a second argument. The argument that I'm going to include is called isDescription, and I'm going to give it a Boolean type. And this value of isDescription cannot be null. Perfect. Now, this argument isDescription is a variable, and it accepts either true or false as its values. All right, so all of our fields are queried within the fragment, so I'm going to look for another field, which is the description of the session. And I talked about directives, right? So directives begin with the symbol @, and I'm looking for the directive include, and within brackets, I'm going to say @include(if $isDescription). Now what this really means is I'm going to include the description field as a part of my query only if the variable isDescription is true. If isDescription is false, then we would not have the description of the session as a part of our query. By doing this, we're basically building a dynamic query which changes based on the variable that has been passed to it. Next step is to pass the variable to our useQuery hook, so I'm going to hardcode isDescription to be true. And based on your client's needs in the UI, this doesn't need to be hardcoded, but just for the demonstration purposes I'm hardcoding it and passing it as the second variable to our useQuery hook. I'm also including description in our UI so we can see that in our Sessions page. Perfect. So let's go ahead and take a look at our Globomantics app. You can see now our Sessions page contains this whole blob of text on each tile, and that is the description of each conference session. And we've displayed that now because we passed true to our include directive. Let's try to change the value to false now. So the expectation now is if the isDescription is false, then the description field will not be included as a part of our query. So let's see if that happens. So I'm going to go back to our conference page, and here we are. Our description is gone now, and it's marked it as undefined because within the UI, I'm still trying to render the description. So I'm going to add a quick check here to see if there is a description that has been returned by the JSON response. If it is the case, then we can go ahead and display it. Our UI is cleaned up too. So this is how we build dynamic queries using directives in GraphQL.

Module Review

Well, that's a wrap. I think we learned quite a bit in this module, and it's probably good to review the concepts we went through. So review time for queries. First things first, we learned about the useQuery hook, which is the backbone for executing queries in GraphQL using the Apollo Client when you're using React. It executes your query, and it returns the JSON response. We then learned about passing arguments to our GraphQL query and also passing variables, which are basically variable arguments. We used variables and arguments to display the conference sessions based on the day, and we also used variables to filter out the conference speakers by the ID. We also looked into error handling that the useQuery hook provides to us. We then learned about fragments, which are the reusable units in GraphQL, and we can reuse them across multiple queries. We then learned about the concept of aliases, where we can pass in multiple arguments to the same fields by just giving it an alias name. And finally, we learned about executing dynamic queries using the concept of directives. In the next module, we're going to learn about mutations, so get set and be ready for writing mutations.

Mutations - Modify and Update GraphQL Data

Module Overview

So far, we've retrieved data using queries from our GraphQL server. It's time now to modify and update our GraphQL data, and to do that, we're going to use what's called mutations. We're going to learn about GraphQL mutations using the Apollo Client. In this module, some of the concepts that we're going to look into are: the useMutation hook to perform mutations. We're going to learn about creating new data. We'll also learn about modifying existing data to our GraphQL server. We'll then track our mutation status. We'll also learn about updating the Apollo cache after a successful mutation. And don't forget, we'll be demoing all these concepts using our Globomantics conference site. All right, let's dive in.

Mutations

Let's take a quick overview of mutations in GraphQL. Mutations in GraphQL can create, update, and delete data. Mutations are very similar to queries, except mutation fields run in series one after the other, whereas queries are executed in parallel by GraphQL. This is because GraphQL is going to assume that the mutation causes a side effect. And to avoid any conflict, the mutation fields run one after the other. Once you're done executing a mutation, we want to make sure that the Apollo cache is updated with the latest data. Most of the time you get this for free, but there are times when you need to manually update the cache, and we'll look at that in just a little bit. Just like the useQuery hook that we used, for mutations, the useMutation is a React hook, which is the API for executing GraphQL mutations in an Apollo application. It makes executing mutations very simple and intuitive. Let's take a look at an example code snippet to understand the useMutation hook. Here we're importing the useMutation hook from the Apollo Client, and we also have the GraphQL function. We've defined an ADD\_TODO function here, and this basically defines a mutation within the GraphQL function just like a query. So the syntax here is very similar to queries, except you use the keyword mutation. And to this ADD\_TODO function, what I'm trying to do is basically add an item with its ID and type. I'm adding a todo item to it. The type is a variable of type string, and we're passing it down to the ADD\_TODO function. Once you've defined the mutation, it's time to execute the mutation, and to do that, we're going to use the useMutation hook. The useMutation hook takes in the ADD\_TODO mutation function as a parameter. So far, it looks very similar to the useQuery hook, but you can notice that there are two objects that we have returned from the useMutation hook. The useMutation hook returns a mutate function. In our case, the mutate function is the ADD\_TODO function, and this is the function that you can call at any time within your code to execute the mutation. So what this means is your mutation is not executed yet. You're going to have to use the ADD\_TODO function within your code to execute it. It also returns an object with fields that represent the current status of the mutation's execution, and that is what we have stored here in data. We're now all set to write a mutation for our Globomantics app in the next clip.

Mutation to Create Session

In this demo, we're going to write a mutation to create a new session for the conference, and we'll also have to make sure that the new session, which we added, is a part of the conference sessions page. Alright, let's get started. We're back into our application here. I've set up the basic UI that we need for our application to submit a session. And to do that, I'm using Formik, a library to create a form factor where we can submit our session along with the details of the session. I haven't written any mutations yet, and this is just the UI setup. If you scroll below, you can see we have a function called AddSession, and this is our component for creating a session. The session form function is then invoked by AddSession, which contains the UI where we call the Formik form. You can notice that our form has fields like title, description, day, level, and we also have a Submit button, and clicking on that button, ideally should submit our form and create a new session. Notice, I've also added a link to our Sessions page that says, Submit a Session, and clicking on this will take us to this new page called sessions/new, where we can then fill out the form to create a new session. We're then adding our new page to the route within the conference page. Now, within our View Sessions page, you can see we have a new text that says, Submit a Session! Clicking on that takes us to our sessions/new page, and this is where you see our form, and the expectation is that the user is going to fill out all this and submit a new session, and that's going to create a new session for us for our conference. First things first, we import the useMutation hook from the Apollo Client, and now we're going to define our mutation. I'm going to call it create session, because that's what we're going to do, and remember, that mutations are also defined within the GraphQL function in Apollo, just like we did for queries. So it's pretty much the same syntax, except we use the keyword mutation instead of the keyword query. I'm going to call our mutation createSession, and we're going to call the mutation createSession that's available to us from our GraphQL server, thanks to auto‑complete right here. Now, first things first, to our createSession mutation, we need to pass in the session information. So the idea here is we are creating a new session, and the session object needs to be passed to it. And this is going to be variable because the user is going to create different sessions each time. It's going to be of type SessionInput, and this can't be null, so we're going to give the exclamation mark here, and we're passing down the session variable to our createSession mutation field. Perfect. Now, every time you run a mutation, you pass to it the variables that you want to be part of the mutation, and the mutation also returns a response back. So I'm going to get the id field and the title field as the response back, and that would be the updated or the new id and the new title that comes back after the mutation is complete. Alright, so our createSession mutation is defined now, and the next step is to execute this mutation. So I'm going to execute this mutation using our useMutation hook, and it's going to return the mutate function, in our case, we're going to call it create, and a data response back, and to the useMutation function, we're going to pass in our mutation that we just wrote, which is CREATE\_SESSION, and with this, our useMutation is called. Now notice that we haven't really executed the mutation yet. What this means is we haven't created a new session yet. The execution happens only when we call the create function. The create function is basically a mutate function that we get back as a response from the useMutation hook, and only when the create function is called, the actual session is created and the mutation occurs. So I'm going to go ahead and call this create function when we submit our form. This is the onSubmit function that we coded for the form. When the user is done filling out the form values, we click on the onSubmit function, and this is where we want to create the session. So I'm going to call the create session as an async function, and to this async function, I'm going to pass in the values, and these values are basically the user‑entered values to the form, which is going to include the title, description, the level, the day of the session, and we're now going to call our create function, and to this create function, we're going to pass in the variables for our mutation. Recall that we've done the same thing for useQuery, but all of that happened at the top level when we were calling the useQuery hook. Here, we pass in the variables to the create function, that is basically the mutate function. And to this, we're passing the session values, and the values are the user‑entered values. Alright, so at this point, we have called our mutation and executed it, and we have defined it as well within the session form. Alright, let's go ahead and check out our Globomantics app to see if we can actually create a new session. I'm clicking on Submit a Session!, and I'm going to give a title for our session. Let's call it My new session. Let's give a description. I'm going to say this is a great session, and the day we're going to pick is Friday, and I'm going to call the level as intermediate. Alright, before I hit Submit, let's take a look at our Network tab to make sure that this actually did work out. So I'm going to hit Submit, and there you go. I see our GraphQL server has been called, and it returned a response 200, which means good news, it worked out. You can scroll down to see how this really looked. So you can see our operation name was createSession, and we've passed in some variables, and the variable is of type session, and it passed in the day, description, level, and title, and all of these are the user‑entered values, and we got back a 200 response, indicating that the mutation worked out well. Now, to check this out, we can go back to our Sessions page. Pardon me. We still don't have the UI hooked up to say that the session was submitted. Now we know we submitted a session for Friday, so I'm going to click on our Friday filter, and I'm going to search for the name of the session we just submitted. And there I see it, My new session, and I see that the My new session was for day Friday, and we also have the level as intermediate, and all of the other fields are null. Perfect! So our mutation worked out well, and we created a new session, and we also verified that it got updated, and we see that on our website. Now, wait a minute. You might be wondering, as a conference website user, can I just go into any website and just submit a session, and the session's approved, and it's part of our schedule? No, we don't want to do that. Ideal conference sites need secure ways to create sessions, and that's not in the scope of this course. The purpose of this course is to learn about queries and mutations, and all about the Apollo Client for React. If you want to further improve your application, the next course that you can take is Securing a GraphQL API with Apollo, and Matt, my friend there, is going to talk about all the cool ways to secure your GraphQL API, and he's going to continue on where we left off. He's going to use the Globomantics conference app and teach you how to secure the Apollo API and the client code, which is why that's not part of our course scope here. So let's go ahead and continue our learning about mutations.

Tracking Mutation Status

So far, we've learned about creating a new session using mutations, we created a new dataset, and we also displayed the new dataset that we created. Did you notice that on our conference website when we submit a new session, we don't really respond to the user that they submitted a session. That's because we're not tracking the mutation status, so let's go ahead and update our code to track the mutation status. Here's where we had called our useMutation hook and we extracted the create mutate function. Now to track the mutation status, we're also going to extract some other items that come back as a response from the useMutation. This is very similar to what we did with the useQuery. We're going to extract the status of the mutation. We're going to extract an object called called and the error object. So as the name suggests, called indicates that the mutation has been executed and muted function has been called. So once you submit the form, it's in the called status. So if called is true, then I'm going to return that the session has been submitted successfully. And similarly, if error is true, then we're going to return that we failed to submit the session. With these changes, our code is going to be a bit more responsive, so we're going to let the user know if they successfully submitted the session or if they failed to submit the session. All right, let's check this out. I'm going to give a title for our session, call it Brand new session, a description, day, and level. I'm going to go ahead and click the Submit button, and there we see our response back that says Session Submitted Successfully! meaning that our mutate function was called successfully and our session has been created, and this is how we track our mutations.

Mutation to Update Featured Speaker

So far, we've written mutations for the sessions, and we created a new session using the useMutation hook. I think it's time to play around a little bit with the speakers. Have you noticed in conferences there are like hundreds of speakers, and there are these five or six speakers who are featured, and they had the star speakers? They're pretty much like the celebrity of the show. Yeah, let's implement the same concept to our Globomantics application. In our Speakers page, I've now added a button that says Featured Speaker with a star. Now, if you do see any speaker with that star in yellow, that means they are a featured speaker, and they are that celebrity we were talking about. All right, so the idea is I want to be able to click on these buttons and basically mark a speaker as featured. And that would require me to write a mutation, and that mutation is going to mark a specific speaker as featured and update that star to yellow. Again, keep in mind, not everybody who's attending a conference would be able to do this; only the admin can do that. And since we don't have login or admin features in our app, anybody who's using this app can do that. As I talked about in earlier clips, this is not in the scope of this course. All right, let's go ahead and look at where I've added this Featured Speaker button and continue writing a mutation to update this. In our Speakers page here, I updated our SpeakerInfo fragment to include the featured field, and that's going to tell us if each speaker is featured or not, and that's basically a Boolean value. And I also updated our UI. In our UI here, I included a button that's going to say Featured Speaker, and if the featured flag is true, the star is going to be golden. If not, our star is not going to be colored. Now it's time to write our mutation and actually make this work. First things first, we have our useMutation imported from the Apollo Client, and I'm going to go ahead and write our mutation. I'm going to define our mutation as const FEATURED\_SPEAKER, and remember to define your mutation within the GraphQL function. In here we're going to use the keyword mutation, and I'm going to call it markFeatured. We're going to use the mutation that comes from our GraphQL API called markFeatured, and thanks to autocomplete, it filled out the arguments for it. So we need a speakerId of type ID, which cannot be null. And also, we need a Boolean called featured, which also cannot be null. We then have to pass these variables to our markFeatured function. We're going to use a dollar sign to pass the speakerId, as well as the featured Boolean. Using this mutation, we can mark a speaker as featured or not, depending on the Boolean value that we give, and the only input that it really needs is the speakerId and the Boolean value for featured. All right, so our mutation is all set and ready, and we're also going to respond back with the speakerId for our mutation. The next step here would be to execute our mutation, and we're going to plug in our useMutation hook. I'm going to do that in our SpeakerList function, right below the useQuery hook. We're going to extract our mutate function, which is going to be called markFeatured, and I'm going to call the useMutation, and to that, we're going to pass our constant FEATURED\_SPEAKER. All right, so our mutation is defined and the useMutation hook has been called, but we haven't yet executed our mutation, and that's going to happen only when we execute our mutate function. In this case, our mutate function is the markFeatured function, and I'm going to call that inside of the onClick within this button, just like we did in our previous mutation. Within the onClick function we're going to make an async call to our markFeatured mutate function, and to the markFeatured function we're going to pass the variables. We do have the speakerId in scope here within this function, so I'm going to pass the speakerId, which has been extracted already. And I'm going to make it simple, and every time they click this button, I'm going to assume that they want to mark this speaker as featured, so I'm just going to pass through to the featured variable. All right, our variables are passed, and when this button's clicked, our mutation's going to be executed. At this point, we're all set to go and check this out in our Speakers page. So we're all set, and our Speakers page has loaded. I'm going to go ahead and click on one of them, and for us to be able to see if the call was successful, let's open our Network tab and maybe click on another one here. I'm going to click on Jerome Parker, and you can see that there was a GraphQL call made there, which was successful, and we see that the markFeatured mutation was called, and the featured flag has been set to true, and the speakerId has been given back as well. Let's go ahead and mark some more speakers as featured, and you can see a couple more calls out there. All right, now let's close the Network tab and refresh our page here to see what happens. There you go. You see that our stars have turned golden for all the three speakers that we just marked as the featured speakers, and that reflects that our data has been updated as well. This might be a good time for you to go over all the speakers and give them a celebrity status and mark them is featured as you please. See you in the next clip.

Updating Cache after Mutation

Let's explore the concept of updating our Apollo cache after a mutation. I know what you're thinking. You're wondering, is there a way to see the updated featured speaker and the gold star without refreshing the page? Of course there is a way to do that. The reason we didn't see an instant gold star was because our Apollo cache was outdated and we had to refresh to get the most up‑to‑date data. When you're mutating just one field, you get this for free with Apollo. All you have to do is when you call your mutation, you have to return back the ID and the mutated field. In the mutation that we wrote here, which was markFeatured, we're just returning back the ID of the speaker. Let's go ahead and include the field that we mutated. In our case, we mutated the field featured, so let's return featured as well in our mutation. By doing this, we're going to get an updated cache after the mutation for free with no other code changes. Let's go and check it out. We're back in our Speakers page here, and I'm going to mark the speaker as featured. And there you go, you see an instant response. Our star is golden now because we got back the mutation response and our Apollo cache was updated. You can keep marking as many speakers as featured, and the gold stars are going to be instant. You can always double‑check and make sure the network calls were successful by opening the Network tab. Mark a few more speakers as featured, and you can see the network calls made right there, and all of them return a status of 200. What's really happening here is our application's UI updates immediately to reflect the changes in the Apollo cache? Well, that answered your question. We were able to update our Apollo cache after running a mutation for a single entity.

Updating Cache after Complex Mutations

So far, we saw how to update our cache after making a simple mutation. The mutation we saw in the previous clip was to mark a speaker as featured. This was a change to one single entity and update to the cache was automatic. But it's not always as simple when you have multiple entity changes. In this clip, we're going to learn about updating the Apollo cache after complex mutations. If a mutation modifies multiple entities, or if it creates or deletes entities, then the Apollo cache is not going to automatically update or reflect the result of your mutation, which means we need to manually update our Apollo cache. And that's what we're going to learn about in this clip. We're back in our Globomantics Sessions page, and I've added another tab here called All Sessions, and this is going to be a simple query that retrieves all our sessions without any complexity. All right, so what I'm going to do right now is submit a session. All right, I'm going to give a title to our session that says MY NEW GRAPHQL SESSION. I'm going to fill in the rest of the details and submit. We've submitted our session successfully. So let's go back to our Sessions Page, and we're in the All Sessions tab. At this point, you're aware that what is displayed is the cached version of our sessions. And after we ran our mutation, let's see if our mutation got updated within our cached version. So I'm going to search for our session, MY NEW GRAPHQL SESSION, and it doesn't seem to like it. It's not available. There are some other GraphQL sessions, but the one that we just created is not a part of this cached list. In situations like this, the only way for the UI to get the updated data would be to hit refresh. And by hitting refresh, we're going to make the call again to the all sessions query and get the updated data with the new session. If you want your UI to be seamless and want instant updates to your cache, then we need to manually update our cache within our code. So let's go ahead and take a look at how we do that. We've added the new query, ALL\_SESSIONS, which is basically a simple query to return all the sessions, and it uses the SessionInfo fragment, and I've also updated our createSession mutation to now return all of the session info instead of just the ID and title. Now, let's go into where we call our useMutation. Here we call our useMutation to create a session, which returns the create, which is a mutate function. Now, to update our cache, I'm going to include another parameter, and I'm going to call what's called the update function. And when the update function is called, it's going to, in turn, call updateSessions. Now what this does is once the mutation is complete, it's going to try to go and update and call the updateSessions function. So we need to now create our updateSessions function. Our updateSessions function is going to take the cache and also the data as a parameter, and this comes from the update function, which useMutation's calling. And in here, the first thing we're going to do is use the keyword cache.modify. And by doing this, we're essentially modifying the existing cache. And the fields that we're going to modify here are the sessions field, because our query is going to return the sessions, and I'm going to initialize an array called existingSessions, and this is going to take in all the existing sessions that we already have in our cache. Oops, I've forgotten to include the arrow function there, so make sure to include the arrow. The next step would be to extract the newSession. Our newSession that we just added is available within the data objects, so I'm going to extract that. Now we're going to use what's called the writeQuery function. So I'm going to call cache.writeQuery. And to this, we're going to pass the query that we're using, which is the ALL\_SESSSIONS query, and we're now going to pass to it the updated data object. So our data is going to now contain the newSessions information, as well as the existing sessions. At this point, our update function is ready. Let's take a quick recap at what we just did. Our update function is invoked once a mutation is complete, and our update function is calling another function called updateSessions, which takes in the cache and the data as parameters. Here, the next step would be to modify our cache so that our new session is included. So what we do here is obtain the current sessions, which is the existingSessions, and then we also obtain the new session and use the writeQuery method to go ahead and update our sessions. Now let's go ahead and take a look at our website. I'm going to now submit a session again. This time, I'm going to call our session Has my cache updated? And let's give it a description, day, let's call it Wednesday, and Level Intermediate. Our session's being submitted successfully, and I'm going to go back to our conference session's page. Let's open our network tab just to double‑check that the query is not being executed again when we click on Sessions. I don't see any queries there, which means we fetched our cached data. And let's look for our new title, I see it right there, Has my cache updated? Yep. So our cache was updated after we created our session, and we didn't really have to refresh the page and the UI automatically showed you the new data. And that demonstrates how to update a cache when you're doing complex mutations creating or deleting data.

Course Summary and Next Steps

Well, we've reached the end of this course, and it's been quite a journey. Let's recap, or revisit, the concepts that we learned in this course. We first learned about the Apollo Client that we can use to consume GraphQL APIs. We worked on setting up the Apollo React Client. We then worked on setting up our Globomantics conference app, which we're going to use throughout the course. We then wrote some introspection queries to analyze our schema in‑depth and understood how introspection can be really useful for clients. We then wrote a number of queries to retrieve data. We displayed this data on our Globomantics app. We got introduced to what's called the useQuery hook that comes out of the box with Apollo React. We used this to write all of our queries and display our conference sessions and speaker information on our website. We finally wrote mutations to update and modify our data. We used the useMutation hook that also comes out of the box with React Apollo. We modified data for our Globomantics app and played around with that. At this point, you're familiar with consuming a GraphQL API and using Apollo as your client. The next step would be to continue your journey with GraphQL and complete the rest of the courses in this path. Feedback is extremely important to us authors. Feel free to ask questions or leave a note on the Discussions tab here on Pluralsight. Leave a star rating so I can understand if you enjoyed this course. If you did enjoy the course, make sure to share this with your team and friends. And I'd like to say thank you. Thank you for spending this hour with me and watching this course. I hope you learned quite a bit. To stay in touch, you can hit me up on Twitter @AdhithiRavi or visit my website adhithiravichandran.com. Again, all the best in your journey in learning GraphQL.