# Hands-on Lab 7: Using GraphQL

**Estimated time: 60 minutes**

## Exercise 1:

Over the last few chapters, we’ve explored how to build React applications that interact with servers using JSON and HTTP APIs. In this chapter, we’re going to explore GraphQL, which is a specific API protocol developed by Facebook that is a natural fit in the React ecosystem.

What is GraphQL? Most literally it means “Graph Query Language”, which may sound familiar if you’ve worked with other query languages like SQL. If your server “speaks” GraphQL, then you can send it a GraphQL query string and expect a GraphQL response. We’ll dive into the particulars soon, but the features of GraphQL make it particularly well-suited for cross-platform applications and large product teams.

To discover why, let’s start with a little show-and-tell.

## Your First GraphQL Query

Typically GraphQL queries are sent using HTTP requests, similar to how we sent API requests in earlier chapters; however, there is usually just one URL endpoint per-server that handles all GraphQL requests.

[GraphQLHub](https://www.graphqlhub.com/)[[1]](#footnote-1) is a GraphQL service that we’ll use throughout this chapter to learn about GraphQL. Its GraphQL endpoint is https://www.graphqlhub.com/graphql, and we issue GraphQL queries using an HTTP POST method.

Fire up a terminal and issue this cURL command:

$ curl -H 'Content-Type:application/graphql' -XPOST https://www.graphqlhub.com/graphql?pretty=true -d '{ hn { topStories(limit: 2) { title url } } }'

{

"data": {

"hn": {

"topStories": [

{

"title": "Bank of Japan Is an Estimated Top 10 Holder in 90% of the Nikkei 225",

"url": "http://www.bloomberg.com/news/articles/2016-04-24/the-tokyo-whale-is-quietly-buying-up-huge-stakes-in-japan-inc"

},

{

"title": "Dropbox as a Git Server",

"url": "http://www.anishathalye.com/2016/04/25/dropbox-as-a-true-git-server/"

}

]

}

}

}

It may take a second to return, but you should see a JSON object describing the title and url of the top stories on [Hacker News](https://news.ycombinator.com/)[[2]](#footnote-2). Congratulations, you’ve just executed your first GraphQL query!

Let’s break down what happened in that cURL. We first set the Content-Type header to application/graphql - this is how the GraphQLHub server knows that we’re sending a GraphQL request, which is a common pattern for many GraphQL servers (we’ll see later on in “Writing Your GraphQL

Server”).

Next we specified a POST to the /graphql?pretty=true endpoint. The /graphql portion is the path, and the pretty query parameters instructs the server to return the data in a human-friendly, indented format (instead of returning the JSON in one line of text).

Finally, the -d argument to our cURL command is how we specify the body of the POST request. For GraphQL requests, the body is often a GraphQL query.

We had to write our request in one line for cURL, but here’s what our query looks like when we expand and indent it properly:

*// one line*

{ hn { topStories(limit: 2) { title url } } }

*// expanded*

{

hn {

topStories(limit: 2) {

title

url

}

}

}

This is a GraphQL query. On the surface it may look similar to JSON, and they do have a tree structure and nested brackets in common, but there are crucial differences in syntax and function.

Notice that the structure of our query is the same structure returned in the JSON response. We specified some properties named hn, topStories, title, and url, and the response object has that exact tree structure - there are no extra or missing entries. This is one of the key features of GraphQL: you request the specific data you want from the server, and no other data is returned implicitly.

It isn’t obvious from this example, but GraphQL not only tracks the properties available to query, but the type of each property as well (“type” as in number, string, boolean, etc). This GraphQL server knows that topStories will be a list of objects consisting of title and url entries, and that title and url are strings. The type system is much more powerful than just strings and objects, and really saves time in the long-run as a product grows more complex.

## Chapter Preview

There are two sides to using GraphQL: as an author of a client or front-end web application, and as an author of a GraphQL server. We’re going to cover both of these aspects in this chapter and the next.

As a GraphQL client, consuming GraphQL is as easy as an HTTP request. We’ll cover the syntax and features of the GraphQL language, as well as design patterns for integrating GraphQL into your JavaScript applications. This is what we’ll cover in this chapter.

As a GraphQL server, using GraphQL is a powerful way to provide a query layer over any data source in your infrastructure (or even third-party APIs). GraphQL is just a standard for a query language, which means you can implement a GraphQL server in any language you like (such as Java, Ruby, or C). We’re going to use Node for our GraphQL server implementation. We’ll cover writing GraphQL servers in the next chapter.

## Consuming GraphQL

If you’re retrieving data from a server using GraphQL - whether it’s with React, another JavaScript library, or a native iOS app - we think of that as a GraphQL “client.” This means you’ll be writing GraphQL queries and sending them up to the server.

Since GraphQL is its own language, we’ll spend this chapter getting you familiar with it and learning to write idiomatic GraphQL. We’ll also cover some mechanics of querying GraphQL servers, including various libraries and starting off with an in-browser IDE: GraphiQL.

## Exploring With GraphiQL

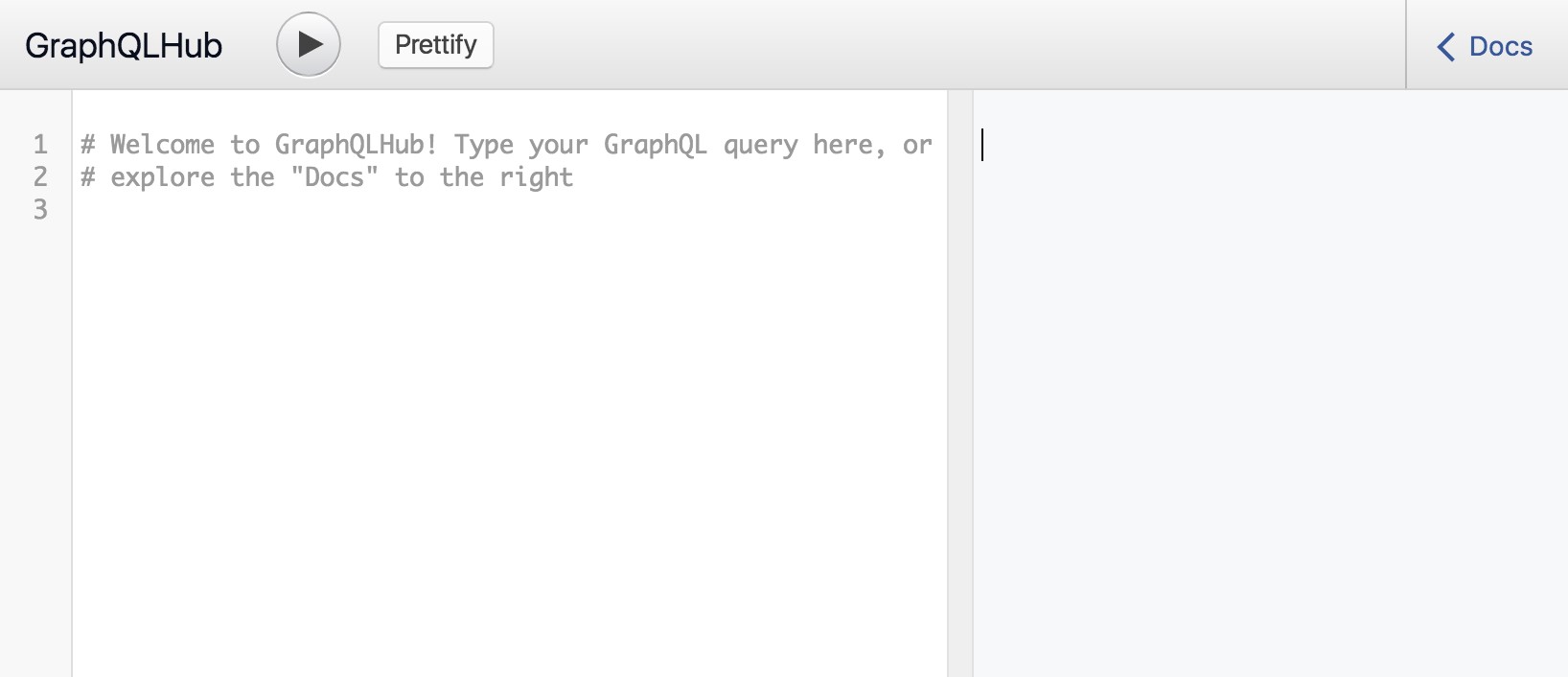
At the start of the chapter we used GraphQLHub with cURL to execute a GraphQL query. This isn’t the only way GraphQLHub provides access to its GraphQL endpoint: it also hosts a visual IDE called [GraphiQL](https://github.com/graphql/graphiql)100. GraphiQL is developed by Facebook and can be used hosted on any GraphQL server with minimal configuration.

You can always issue GraphQL requests using tools like cURL or any language that supports HTTP requests, but GraphiQL is particularly helpful while you become familiar with a particular GraphQL server or GraphQL in general. It provides type-ahead support for errors or suggestions, searchable documentation (generated dynamically using the GraphQL introspection queries), and a JSON viewer that supports code folding and syntax highlighting.

Head to [https://graphqlhub.com/playground1](https://graphqlhub.com/playground)01 and get your first look at GraphiQL:

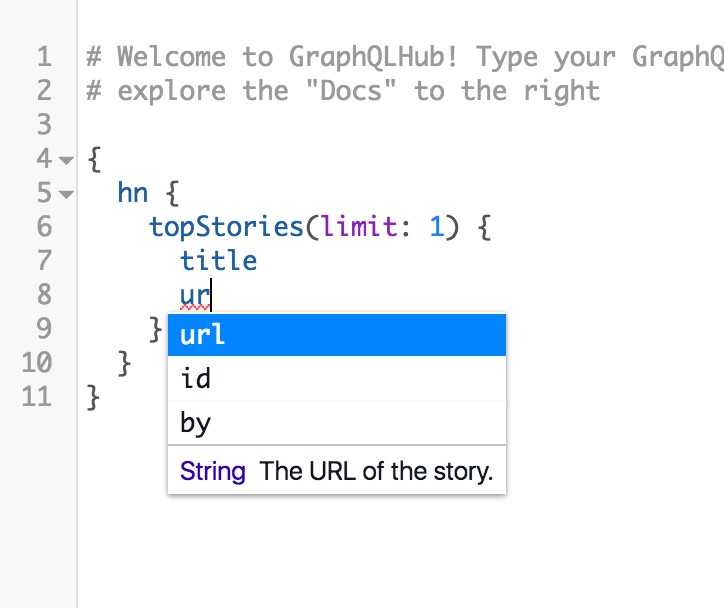
100<https://github.com/graphql/graphiql>

101<https://graphqlhub.com/playground>



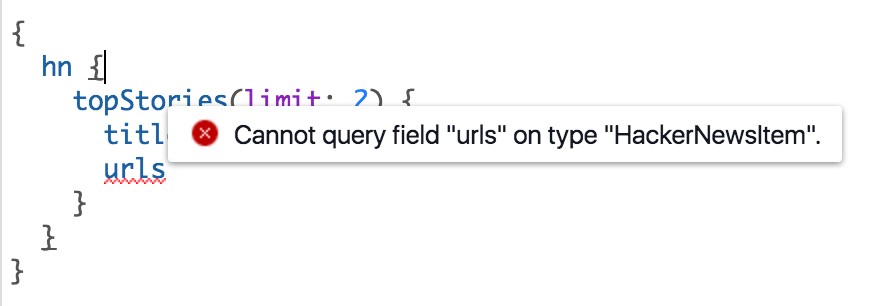
Empty GraphiQL

Not much going on yet - go ahead and enter the GraphQL query we cURL’d from earlier:



GraphiQL query

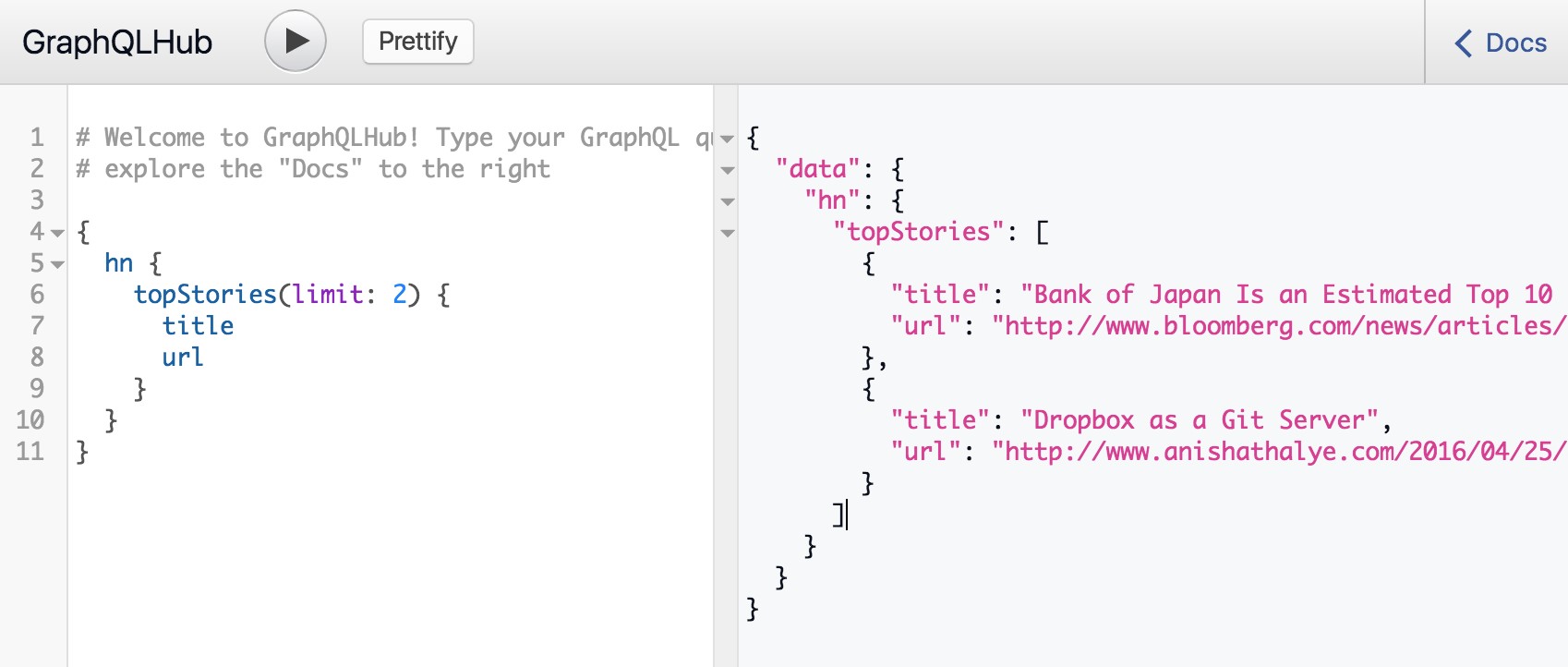
As you type the query, you’ll notice the helpful typeahead support. If you make mistakes, such as entering a field that doesn’t exist, GraphiQL will warn you immediately:



GraphiQL error

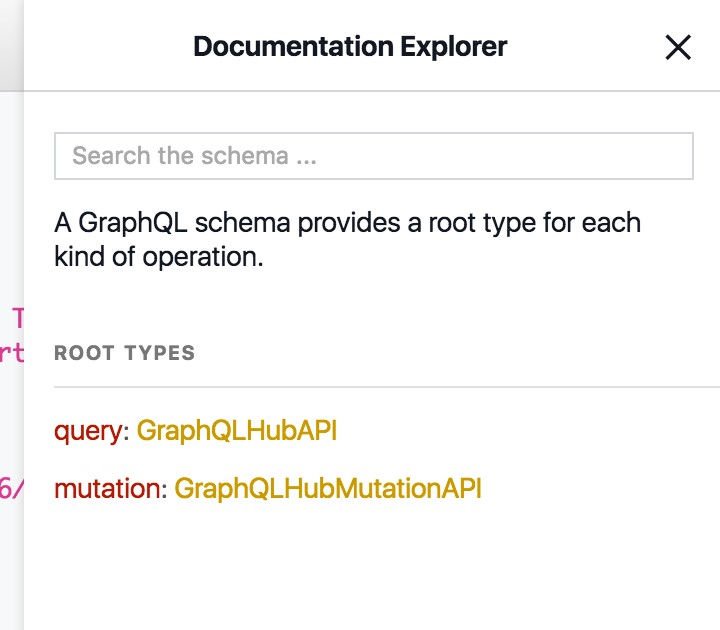
This is a great example of GraphQL’s type system at work. GraphiQL knows what fields and types exist, and in this case the field urls does not exist on the HackerNewsItem type. We’ll explore how types get their fields and names later on.

Hit the “Play” button in the top navigation bar to execute your query. You’ll see the new data appear in the right pane:



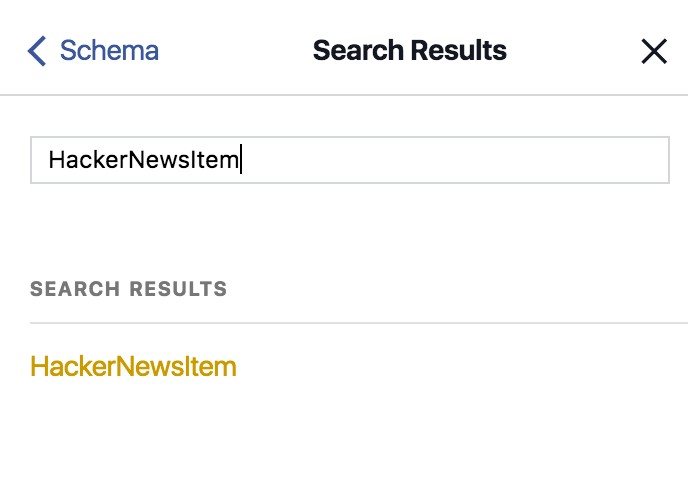
GraphiQL data

See that “Docs” button in the top right corner of GraphiQL? Give that a click to expand the full documentation browser:



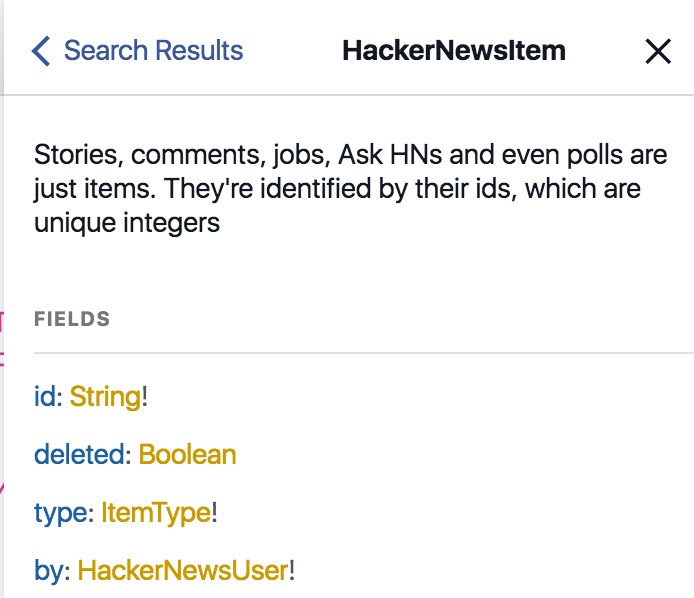
GraphiQL docs

Feel free to click around and choose-your-own-adventure, but eventually come back to the page in the screenshot (the top-level page) and search that HackerNewsItem type we ran into trouble with earlier:

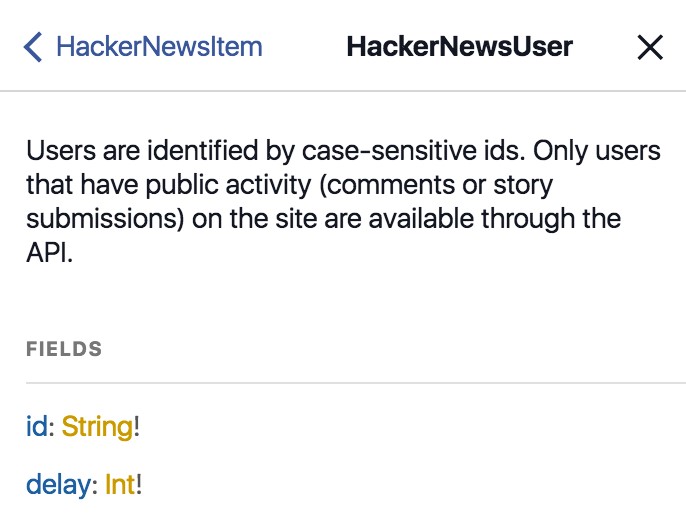


GraphiQL search

Click the matching entry. This takes you to the documentation describing the HackerNewsItem type, which includes a description (written by a human, not generated) and a list of all the fields on the type. You can click the fields and their types to find out more information:



GraphiQL HackerNewsItem



GraphiQL HackerNewsUser

As you can see, the by field of HackerNewsItem has the type HackerNewsUser, which has its own set of fields and links. Let’s change our query to grab the information about the author:



GraphiQL query with HackerNewsUser

Notice how the by field now shows up both in our query and in the final data - ta da!

Keep GraphQLHub’s GraphiQL open in a tab as we start to dig into the mechanics of GraphQL.

## GraphQL Syntax 101

Let’s dig into the semantics of GraphQL. We’ve used some terms like “query”, “field”, and “type”, but we haven’t properly defined them yet, and there are still a few more to cover before we delve further into our work. For a complete and formal examination of these topics, you can always refer to the [GraphQL specification](https://facebook.github.io/graphql/)[[3]](#footnote-3).

The entire string you send to a GraphQL server is called a document. A document can have one or more operations - so far our example documents have just only a query operation, but you can also send mutation operations.

A query operation is read-only - when you send a query, you’re asking the server to give you some data. A mutation is intended to be a write followed by a fetch; in other words, “Change this data, and then give me some other data.” We’ll explore mutations more in a bit, but they use the same type system and have the same syntax as queries. Here’s an example of a document with just one query:

query getTopTwoStories {

hn {

topStories(limit: 2) {

title

url

}

}

}

Note that we have prefixed our original query with query getTopTwoStories, which is the full and formal way to specify an operation within a document. First we declare the type of operation (query or mutation) and then the name of the operation (getTopTwoStories).

If your GraphQL document contains just one operation, you can omit the formal declaration and the GraphQL will assume you mean a query:

{

hn {

topStories(limit: 2) {

title

url

}

}

}

In the case that your document has multiple operations, you need to give each of them a unique name. Here’s an example of a document with several operations:

query getTopTwoStories {

hn {

topStories(limit: 2) {

title

url

}

}

}

mutation upvoteStory {

hn {

upvoteStory(id: "11565911") {

id

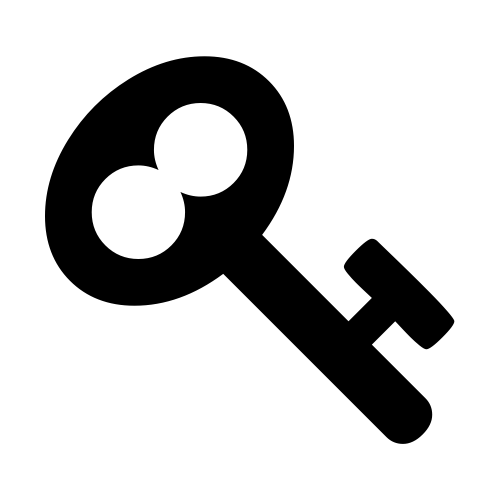
score

}

}

}

Generally you won’t be sending multiple operations to the server, as the GraphQL specification states a server can only run one operation per document.

Multiple operations are allowed in a document for advanced performance optimization[s](https://github.com/facebook/graphql/issues/29#issuecomment-118994619)[[4]](#footnote-4)detailed by Facebook.

So that’s documents and operations, now let’s dig into a typical query. An operation is composed of selections, which are generally fields. Each field in GraphQL represents a piece of data, which can either be an irreducible scalar type (defined below) or a more complex type composed of yet more scalars and complex types.

In our previous examples, title and url are scalar fields (as string is a scalar type), and hn and topStories are complex types.

A unique trait of GraphQL is that you must specify your selection until it is entirely composed of scalar types. In other words, this query is invalid because hn and topStories are complex types and the query does not end in any scalar fields:

{

hn {

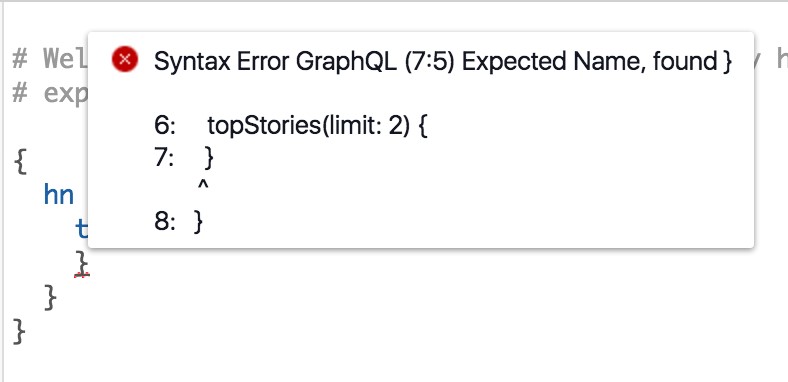
topStories(limit: 2) {

}

}

}

If you try this in GraphiQL, it will tell you eagerly that it is invalid:



GraphiQL error without scalar

More philosophically, this means that GraphQL queries must be unambiguous and reinforces the concept that GraphQL is a protocol where you fetch only the data you demand.

Scalar types include Int, Float, String, Boolean and ID (coerced to a string). GraphQL provides ways of composing these scalars into more complex types using Object, Interface, Union, Enum, and List types. We’ll go into each of those later, but it should be intuitive that they allow you to compose different scalar (and complex) types to create powerful type hierarchies.

Additionally, fields can have arguments. It’s useful to think of all fields as being functions, and some of them happen to take arguments like functions in other programming languages. Arguments are declared between parenthesis after the name of a field, are unordered, and can even be optional.

In our previous example, limit is an argument to topStories:

{

hn {

topStories(limit: 10) {

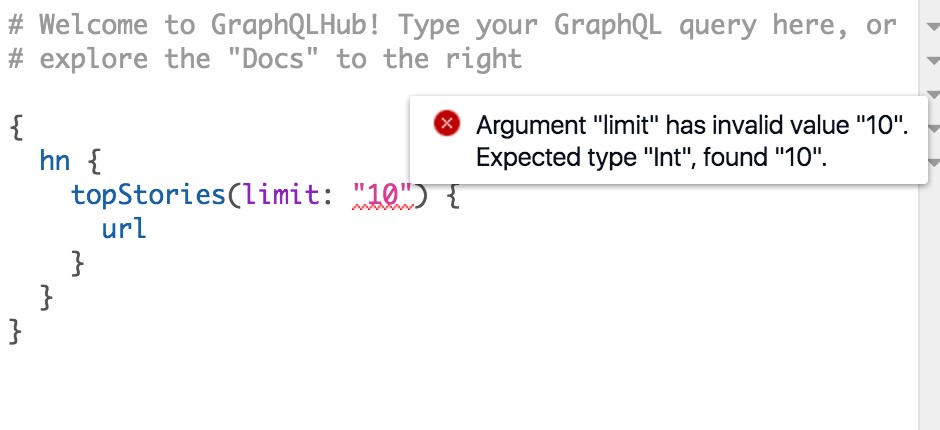
url

}

}

}

Arguments are also typed in the same way as fields. If we try to use a string, GraphiQL shows us the error in our ways:



GraphiQL error argument type

It turns out that limit is actually an optional argument for this GraphQL server, and omitting it is still a perfectly valid query:

{

hn {

topStories {

url

}

}

}

The arguments to a GraphQL field can also be complex objects, referred to as input objects. These are not just the string or numeric scalars we’ve shown, but are arbitrarily deeply nested maps of keys and values.

Here’s an example where the argument storyData takes an input object which has a url property:

{

hn {

createStory(storyData: { url: "http://fullstackreact.com" }) {

url

}

}

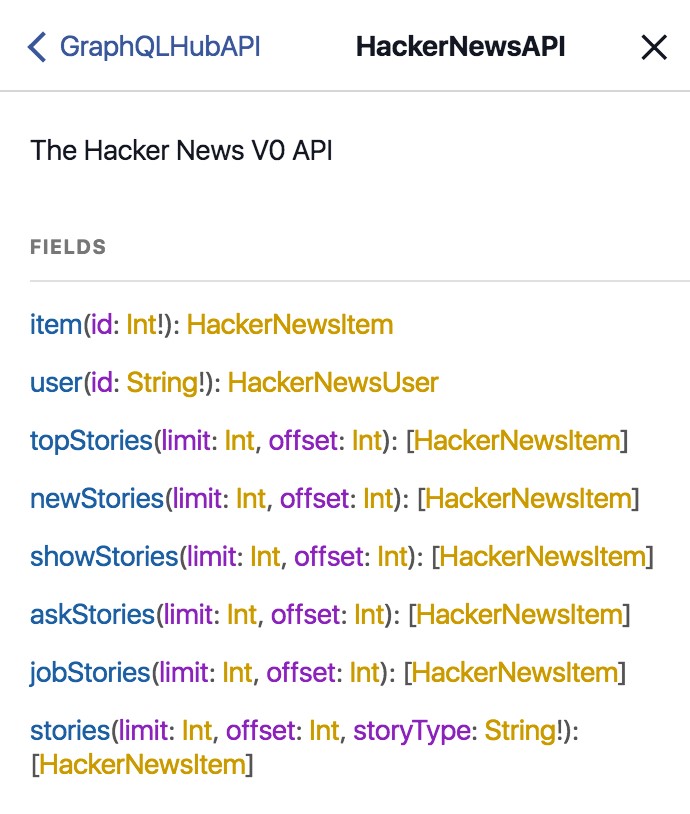
}

The collection of fields of a GraphQL server is called its schema. Tools like GraphiQL can download the entire schema (we’ll show how to do that later) and use that for auto-complete and other functionality.

## Complex Types

We’ve discussed scalars but only alluded to complex types, though we’ve been using them in our examples. The hn and topStories fields are examples of Object and List type fields, respectively.

In GraphiQL we can explore their exact types - search HackerNewsAPI to see details about hn:



HackerNewsAPI type

### Unions

What do you do if your field should actually be more than one type? For example, if your schema has some kind of universal search functionality, it’s likely that it will return many different types.

So far we’ve only seen examples where each field is one type, either a scalar or a complex object how would we handle something like search?

GraphQL provides a few mechanisms for this use-case. First is unions, which allow you to define a new type that is one of a list of other types. This is example of unions comes straight from the [GraphQL spec](https://facebook.github.io/graphql/#sec-Unions)[[5]](#footnote-5):

union SearchResult = Photo | Person

type Person {

name: **String**

age: **Int**

}

type Photo {

height: **Int**

width: **Int**

}

type SearchQuery {

firstSearchResult: **SearchResult**

}

This syntax look unfamiliar? It’s an informal variant of pseudo-code used by the GraphQL spec to describe GraphQL schemas - we won’t be writing any code using this format, it’s purely to make it easier to describe GraphQL types independent of server code.

In that example, the SearchQuery type has a firstSearchResult field which may either be a Photo or Person type. The types in a union don’t have to be objects - they can be scalars, other unions, or even a mix.

### Fragments

If you look closer, you’ll notice there are no fields in common between Person and Photo. How do we write a GraphQL query which handles both cases? In other words, if our search returns a Photo, how do we know to return the height field?

This is where fragments come into play. Fragments allow you to group sets of fields, independent of type, and re-use them throughout your query. Here’s what a query with fragments could look like for the above schema:

{

firstSearchResult {

... on Person {

name

}

... on Photo {

height

}

}

}

The ... on Person bit is referred to as an inline fragment. In plain-English we could read this as, “If the firstSearchResult is a Person, then return the name; if it’s a Photo, then return the height.”

Fragments don’t have to be inline; they can be named and re-used throughout the document. We could rewrite the above example using named fragments:

{

firstSearchResult {

... searchPerson

... searchPhoto

}

}

fragment searchPerson on Person {

name

}

fragment searchPhoto on Photo {

height

}

This would allow us to use searchPerson in other parts of the query without duplicating the same inline fragment everywhere.

### Interfaces

In addition to unions, GraphQL also supports interfaces, which you might be familiar with from programming languages like Java. In GraphQL, if an object type implements an interface, then the GraphQL server enforces that the type will have all the fields the interface requires. A GraphQL type that implements an interface may also have its own fields that are not specified by the interface, and a single GraphQL type can implement multiple interfaces.

To continue the search example, you can imagine our search engine having this type of schema:

**interface** Searchable {

searchResultPreview: **String**

}

type Person **implements** Searchable {

searchResultPreview: **String**

name: **String**

age: **Int**

}

type Photo **implements** Searchable {

searchResultPreview: **String**

height: **Int**

width: **Int**

}

type SearchQuery {

firstSearchResult: **Searchable**

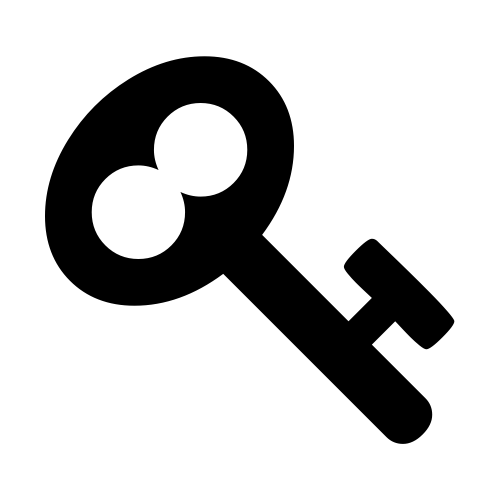
}

Let’s break this down. Our firstSearchResult is now guaranteed to return a type that implements Searchable. Because this otherwise unknown type implements Searchable,

These are the primitives of GraphQL - operations, types (scalar and complex), and fields - and we can use them to compose higher-order patterns.

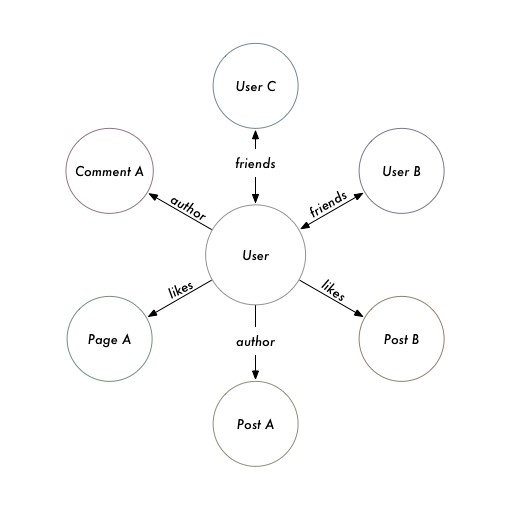
## Exploring a Graph

We’ve explored the “QL” of GraphQL, but haven’t touched too much on the “Graph” part.

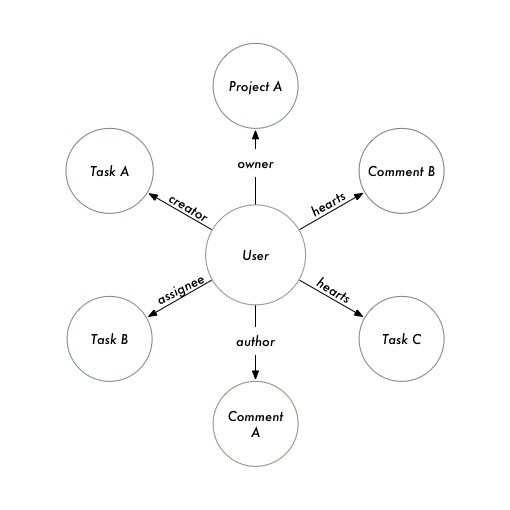
When we say “Graph”, we don’t mean in the sense of a bar chart or other data visualizations - we mean a graph in the more mathematical sens[e](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics))[[6]](#footnote-6) .

A graph is composed of a set of objects that are linked together. Each object is called a node, and the link between a pair of objects is called an edge. You might not be used to thinking about your product’s data with this vocabulary, but it is surprisingly representative of most applications.

Let’s consider the graph of a Facebook user:



Facebook Graph But also consider the graph of a productivity application like Asana:



Asana Graph

Even though Asana is not really a social network, the product’s data still forms a graph.

Note that just because your product’s data resembles a graph structure doesn’t mean you need to use a graph database: Facebook and Asana’s underlying datastore is often MySQL, which doesn’t natively support graph operations. Any data store, whether a relational database, key-value store, or document store, can be exposed as a graph in GraphQL.

The ideal GraphQL schema closely models the shape and terms of a mathematical graph. Here’s a preview of the type of graph-like schema we’ll be exploring:

{

viewer {

id

name

likes {

edges {

cursor 8 node {

id

name

}

}

}

}

node(id: "123123") {

id

name

}

}

Adding these extra layers of fields (edges, node, etc) may seem like over-engineering when coming from something like REST. However, these patterns emerged from building Facebook and provide a foundation as your product grows and adds new features.

The patterns we’re going to describe in the next few sections are derived from the [GraphQL patterns required by Relay](https://facebook.github.io/relay/docs/graphql-relay-specification.html)[[7]](#footnote-7). Even if you don’t use Relay, this way of thinking of your GraphQL schema will prevent you from “fighting the framework” and it wouldn’t be surprising if future GraphQL front-end libraries beyond Relay continue to embrace them.

## Graph Nodes

When querying a graph, you generally need to start your query with a node. Your app is either fetching fields directly on a node or finding out about what connections it has.

For example, the querying “the current user’s feed” in Facebook would start with the current user’s node, and explore all “feed item” nodes connected to it.

In idiomatic GraphQL, you generally define a simple Node interface like this:

**interface** Node {

id: **ID**

}

So anything that implements this interface is expected to have an id field - that’s it. Remember from earlier that the ID scalar type is casted to a string. If you’re coming from the REST world, this is sort of like when you can query a resource using a URL like /$nouns/:id.

One key difference that idiomatic GraphQL and other protocols is all of your node IDs should be globally unique. In other words, it would be invalid to have a User with an ID of “1” and a Photo with an ID of “1”, as the IDs would collide.

The reason behind this is that you should also expose a top-level field that lets you query arbitrary nodes by ID, something like this:

{

node(id: "the\_id") {

id

}

}

If you have IDs that collide across types, it’s impossible to determine what to return from that field. If you’re using a relational database, this might be puzzling because by default primary keys are only guaranteed to be unique per-table.

A common technique is to prefix your database IDs with a string that corresponds to the type and makes the IDs unique again. In pseudo-code:

**const** findNode = (id) => {

**const** [ prefix, dbId ] = id.split(":");

**if** (prefix === 'users') {

**return** database.usersTable.find(dbId);

}

**else if** ( ... ) {

}

};

**const** getUser = (id) => {

**let** user = database.usersTable.find(id);

user.id = `users:**${**user.id**}**`;

**return** user;

};

Whenever our GraphQL server returns a user (via getUser), it changes the database ID to make it unique. Then when we lookup a node (via findNode), we read the “scope” of the ID and act appropriately. This also highlights a benefit of why IDs are strings in GraphQL, because they allow for readable changes like these.

Why do we want the ability to query any node using the node(id:) field? It makes it easier for our web applications to “refresh” stale data without having to keep track of where a piece of data comes from in the schema. If we know a todo-list item changed it’s state, our app should be able to look-up the latest state from the server without knowing what project or group it belongs to.

Generally our apps don’t query global node IDs from the start - how do we describe the “current user’s node” in GraphQL? It turns out the viewer pattern comes in handy.

## Viewer

In addition to the node(id:) field, it’s very helpful if your GraphQL schema has a top-level viewer field. The “viewer” represents the current user and the connections to that user. At the schema-level, the viewer’s type should implement the Node interface like any other node type.

If you’re coming from REST, this either happens implicitly or all of your endpoints are prefixed being with a path like /users/me/$resources. GraphQL tends to prefer explicit behaviors, which is why the viewer field tends to exist.

Imagine we’re writing a Slack app with GraphQL. Everything is viewer-centric in a messaging app (“what messages do I have”, “what channels am I subscribed to”, etc), so our queries would look something like this:

{

viewer {

id

messages {

participants {

id

name

}

unreadCount

}

channels {

name

unreadCount

}

}

}

If we didn’t have the top-level viewer field on its own, we’d either have to make two server calls (“get me the ID of the current user, and then get me the messages for that user”) or make top-level messages and channels fields that implicitly return the data for the current user.

When we implement our own GraphQL server later on, we’ll see that using a viewer field also makes it easier to implement authorization logic (i.e. prevent one user from seeing another user’s messages).

## Graph Connections and Edges

In that last example of a viewer-centric query, we had two fields that you can think of as connections to sets of other nodes (messages and channels). For simple and small sets of data, we could just return an array of all these nodes - but what happens if the data set is very large? If we were loading something like posts of Reddit, we definitely couldn’t fit them into one array.

The usual way to load large sets of data is pagination. Many APIs will let you pass some kind of query parameters like ?page=3 or ?limit=25&offset=50 to iterate over a bigger list. This works well for apps where the data is relatively stable, but can accidentally lead to a poor experience in apps where data updates in near-real-time: something like Twitter’s feed is may add new tweets while the user is browsing, which will throw off the offset calculations and lead to duplicated data when loading new pages.

Idiomatic GraphQL takes a strong opinion on how to solve that problem using cursor-based pagination. Instead of using pages or limits and offsets, GraphQL requests pass cursors (usually strings) to specify the location in the list to load next.

In plain-English, a GraphQL request retrieves an initial set of nodes and each node is returned with a unique cursor; when the app needs to load more data, it sends a new request equivalent to, “Give me the 10 nodes after cursor XYZ.”

Let’s try using a GraphQL endpoint that supports cursors. Take a look at this query that you can send to GraphQLHub:

{

hn2 {

nodeFromHnId(id:"dhouston", isUserId:**true**) {

... on HackerNewsV2User {

submitted(first: 2) {

pageInfo {

hasNextPage

hasPreviousPage

startCursor

endCursor

}

edges {

cursor

node {

id

... on HackerNewsV2Comment {

text

}

}

}

}

}

}

}

}

Pretty big query there, let’s break it down - first we get our initial node using nodeFromHnId (which retrieves the node for [Drew Houston](https://en.wikipedia.org/wiki/Drew_Houston)[[8]](#footnote-8)’s Hacker News account). We then grab the first two nodes in the submitted connection.

A GraphQL connection has two fields, pageInfo and edges. pageInfo is metadata about that particular “page” (remember that this is more of a moving “window” than a page). Your front-end code can use this metadata to determine when and how to load more information - for example, if hasNextPage is true then can show the appropriate button to load more items. The edges field is a list of the actual nodes. Each entry in edges contains the cursor for that node, as well as the node itself.

Notice that the node id and cursor are separate fields - in some systems it may be appropriate to use id as part of the cursor (such as if your identifiers are atomically incrementing integers), but others may prefer to make cursor a function of timestamp, offset, or both. In general, cursors are intended to be opaque strings, and may become invalid after a certain period of time (in the case that the backend is caching search results temporarily).

Let’s say this is the data returned by that query:

{

"nodeFromHnId": { "submitted": {

"pageInfo": {

"hasNextPage": true,

"hasPreviousPage": false,

"startCursor": "YXJyYXljb25uZWN0aW9uOjE=",

"endCursor": "YXJyYXljb25uZWN0aW9uOjI="

},

"edges": [

{

"cursor": "YXJyYXljb25uZWN0aW9uOjE=",

"node": {

"id": "aXRlbTo1MzgxNjk0",

"text": "it's not going anywhere :)<p>(actually, come work on it: <a\

href=\"https://www.dropbox.com/jobs\" rel=\"nofollow\">https://www.dropbox.com/\

jobs</a> :))"

}

},

{

"cursor": "YXJyYXljb25uZWN0aW9uOjI=",

"node": {

"id": "aXRlbTo0NjgxMzY2",

"text": "yes we are :)"

}

}

]

}

}

}

Take a look at how some of the fields match up - the first cursor in edges matches the startCursor, as well as the endCursor matching the last node’s cursor. Our front-end code could use endCursor to construct the query to fetch the next set of data using the after argument:

{ hn2 {

nodeFromHnId(id:"dhouston", isUserId:true) {

... on HackerNewsV2User { submitted(first: 2, after: "YXJyYXljb25uZWN0aW9uOjI=") {

pageInfo {

hasNextPage hasPreviousPage startCursor endCursor

} edges { cursor node {

id

... on HackerNewsV2Comment { text

}

}

}

}

}

}

}

}

The other arguments that exist on connections are before and after (which accept cursors), and first and last (which accepts integers).

The cursor pattern may come off as verbose if you’re used to pagination in REST, but give it five minutes and explore the Hacker News pagination API we demonstrated. Cursors are robust to realtime updates and allow for more reusable app-level code when loading nodes. When we implement our own GraphQL server in a bit, we’ll show how to implement this kind of schema and you’ll see it isn’t as daunting as it may initially appear.

## Mutations

When we first introduced operations, we mentioned that mutation exists alongside the read-only query operation. Most apps will need a way to write data to the server, which is the intended use for mutations.

With REST-like protocols, mutations are generally occur with POST, PUT, and DELETE HTTP requests. In that sense, both GraphQL and REST try to separate read-only requests from writes. So what do we gain with GraphQL? Because GraphQL mutations leverage GraphQL’s type system, you can declare the data you want returned following your mutation.

For example, you can try this mutation on GraphQLHub to edit an in-memory key value store:

mutation {

keyValue\_setValue(input: {

clientMutationId: "browser", id: "this-is-a-key", value: "this is a value"

}) {

item {

value

id

}

}

}

The mutation field here is keyValue\_setValue, and it takes an input argument that gives information what key and value to set. But we also get to pick and choose the fields returned by the mutation, namely the item and whatever set of fields we want from that. If you run that request, you’ll get back this sort of payload:

{

"data": {

"keyValue\_setValue": {

"item": {

"value": "some value",

"id": "someKey"

}

}

}

}

In a REST world, we would be stuck with whatever data the request returns, and as our product evolves we may have to make many changes to that payload on the client and server. Using GraphQL means that our server and client will be more resilient and flexible in the future.

Other than requiring specifying the mutation operation type, everything about mutations is normal GraphQL: you have types, fields, and arguments. As we’ll see later on when we implement our own GraphQL schema, implementing them on the server is similar as well.

## Subscriptions

We’ve discussed the two main types of GraphQL operations, query and mutation, but there is a third type of operation currently in development: subscription. The use-case of subscriptions is to handle the kinds of real-time updates seen in apps like Twitter and Facebook, where the number of likes or comments on an item will update without manual refreshing by the user.

This provides a great user experience, but is often complicated to implement on a technical level. GraphQL takes the opinion that the server should publish the set of events that it’s possible to subscribe to (such as new likes to a post) and clients can opt-in to subscribing to them.

Check out the example subscription that Facebook gives in [their documentation](http://graphql.org/blog/subscriptions-in-graphql-and-relay/)[[9]](#footnote-9):

input StoryLikeSubscribeInput {

storyId: **string**

clientSubscriptionId: **string**

}

subscription StoryLikeSubscription($input: **StoryLikeSubscribeInput**) {

storyLikeSubscribe(input: **$input**) {

story {

likers { count }

likeSentence { text }

}

}

}

Issuing a GraphQL request with this subscription essentially tells the server, “Hey, here’s the data I want whenever a StoryLikeSubscription occurs, and here’s my clientSubscriptionId so you know where to find me.” Note that the use of clientSubscriptionId or any details about what subscription operations should look like is not specific by GraphQL; it merely reserves the subscription operation type as an acceptable and defers to each application on how to handle real-time updates.

The mechanics of how the clients subscribe to updates are outside of the scope of GraphQL -

Facebook mentions using [MQTT](https://en.wikipedia.org/wiki/MQTT)[[10]](#footnote-10) with the clientSubscriptionId, but other possibilities include [WebSockets](https://en.wikipedia.org/wiki/WebSocket)[[11]](#footnote-11), [Server-Sent Events](https://en.wikipedia.org/wiki/Server-sent_events)[[12]](#footnote-12), or any of a number of other mechanisms. In pseudo-code, the process looks like:

**var** clientSubscriptionId = generateSubscriptionId();

*// this "channel" could be WebSockets, MQTT, etc*

connectToRealtimeChannel(clientSubscriptionId, (newData) => {});

*// send the GraphQL request to tell the server to start sending updates*

sendGraphQLSubscription(clientSubscriptionId);

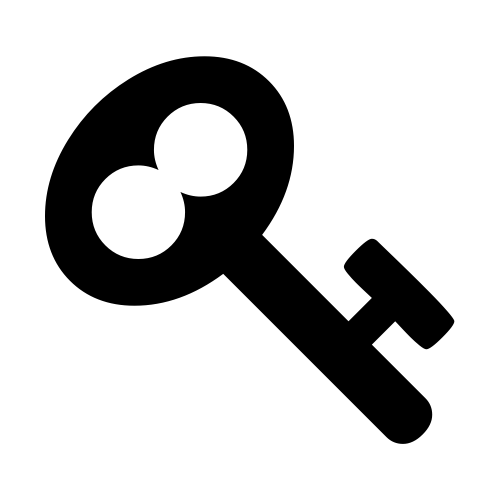
The way GraphQL has decided to implement subscriptions, where a server allows a finite list of possible events, is different than the way other frameworks like Meteor handle updates, where all data is subscribable by default. As Facebook details in [their writing](http://graphql.org/blog/subscriptions-in-graphql-and-relay/#why-not-live-queries)[[13]](#footnote-13), that type of system is generally very difficult to engineer, especially at scale.

## GraphQL With JavaScript

So far we’ve been using cURL and GraphiQL for all of our GraphQL queries, but at the end of the day we’re going to be writing JavaScript web apps. How do we send GraphQL requests in the browser?

Well, you can use any HTTP library you like - jQuery’s AJAX methods will work, or even raw XmlHttpRequests, which enables you to use GraphQL in older non-ES2015 apps. But because we’re examining how modern JavaScript apps work in the React ecosystem, we’re going to examine ES2015 fetch.

Fire up Chrome, open up [the GraphQLHub website1](https://www.graphqlhub.com/)13 and open a JavaScript debugger.

You can open the Chrome DevTools JavaScript Debugger by clicking the Chrome “hamburger” icon and picking More Tools > Developer Tools or by right-clicking on the page, pick Inspect, and then clicking on the Console tab.

Modern versions of Chrome support fetch out of the box, which makes it handy for prototyping, but you can also use any other tooling that supports poly-filling fetch. Give this code a shot:

**var** query = ' { graphQLHub } ';

**var** options = {

method: 'POST',

body: query,

headers: {

'content-type': 'application/graphql'

}

};

fetch('https://graphqlhub.com/graphql', options).then((res) => {

**return** res.json();

}).then((data) => {

console.log(JSON.stringify(data, **null**, 2));

});

The configuration here should look similar to our settings for cURL. We use a POST method, set the appropriate content-type header, and then use our GraphQL query string as the request body.

Give your code a moment and you should see this output:

{

"data": {

"graphQLHub": "Use GraphQLHub to explore popular APIs with GraphQL! Created \

by Clay Allsopp @clayallsopp"

}

}

Congratulations, you just ran a GraphQL query with JavaScript! Because GraphQL requests are Just HTTP at the end of the day, you can incrementally move your API calls over to GraphQL, it doesn’t have to be done in one big-bang.

Making API calls is usually a fairly low-level operation, though - how can it integrate into a larger app?

## GraphQL With React

This book is all about React, so it’s about time we integrate GraphQL with React, right? Almost!

The most promising way of using GraphQL and React is Relay, to which we’re dedicating an entire chapter. Relay automates many of the best practices for React/GraphQL applications, such as caching, cache-busting, and batching. It would be a tall-order to cover the mechanics of how Relay does these things, and ultimately using Relay is the better solution than writing your own.

But adopting Relay may have more friction in an existing app, so it’s worthwhile to discuss a few techniques for adding GraphQL to an existing React app.

If you’re using Redux, you can probably swap out your REST or other API calls with GraphQL calls using the fetch technique we showed earlier. You won’t get the colocated queries API that Relay or other GraphQL-specific libraries provide, but GraphQL’s benefits (such as development experience and testability) will still shine.

There are burgeoning alternatives to Relay as well. [Apollo](http://apollostack.com/)[[14]](#footnote-14) is a collection of projects including [react-apollo](http://docs.apollostack.com/apollo-client/react.html)[[15]](#footnote-15). react-apollo allows you to colocate views and their GraphQL queries in a manner similar to Relay, but uses Redux under-the-hood to store your GraphQL cache and data. Here’s an example of a simple Apollo component:

**class** AboutGraphQLHub **extends** React.Component {

render() {

**return** <div>{ **this**.props.about.graphQLHub }</div>;

}

}

**const** mapQueriesToProps = () => {

**return** {

about : {

query: '{ graphQLHub }'

}

};

};

**const** ConnectedAboutGraphQLHub = connect({

mapQueriesToProps

})(AboutGraphQLHub);

Building upon Redux means you can more easily integrate it into an existing Redux store like any other middleware or reducer:

**import** ApolloClient from 'apollo-client';

**import** { createStore, combineReducers, applyMiddleware } from 'redux';

**const** client = **new** ApolloClient();

**const** store = createStore(

combineReducers({

apollo: client.reducer(),

*// other reducers here*

}),

applyMiddleware(client.middleware())

);

1. <https://www.graphqlhub.com/> [↑](#footnote-ref-1)
2. [https://news.ycombinator.com](https://news.ycombinator.com/) [↑](#footnote-ref-2)
3. <https://facebook.github.io/graphql/> [↑](#footnote-ref-3)
4. <https://github.com/facebook/graphql/issues/29#issuecomment-118994619> [↑](#footnote-ref-4)
5. <https://facebook.github.io/graphql/#sec-Unions> [↑](#footnote-ref-5)
6. <https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)> [↑](#footnote-ref-6)
7. <https://facebook.github.io/relay/docs/graphql-relay-specification.html> [↑](#footnote-ref-7)
8. <https://en.wikipedia.org/wiki/Drew_Houston> [↑](#footnote-ref-8)
9. <http://graphql.org/blog/subscriptions-in-graphql-and-relay/> [↑](#footnote-ref-9)
10. <https://en.wikipedia.org/wiki/MQTT> [↑](#footnote-ref-10)
11. <https://en.wikipedia.org/wiki/WebSocket> [↑](#footnote-ref-11)
12. <https://en.wikipedia.org/wiki/Server-sent_events> [↑](#footnote-ref-12)
13. <http://graphql.org/blog/subscriptions-in-graphql-and-relay/#why-not-live-queries>113<https://www.graphqlhub.com/> [↑](#footnote-ref-13)
14. [http://apollostack.com](http://apollostack.com/) [↑](#footnote-ref-14)
15. <http://docs.apollostack.com/apollo-client/react.html> [↑](#footnote-ref-15)