# Using GraphQL

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[93] 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/g\
raphql?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 Ni\
kkei 225",
          "url": "http://www.bloomberg.com/news/articles/2016-04-24/the-tokyo-wh\
ale-is-quietly-buying-up-huge-stakes-in-japan-inc"
        },
        {
          "title": "Dropbox as a Git Server",
```

[93] https://www.graphqlhub.com/



```
          "url": "http://www.anishathalye.com/2016/04/25/dropbox-as-a-true-git-s\
erver/"
        }
      ]
    }
  }
}
```

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[94]. 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:

```
1  // one line
2  { hn { topStories(limit: 2) { title url } } }
3
4  // expanded
5  {
6    hn {
7      topStories(limit: 2) {
8        title
9        url
10     }
11   }
12 }
```

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

[94] https://news.ycombinator.com

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.

## GraphQL Benefits

Now that we've seen a bit of GraphQL in action, you may be wondering, "why anyone would prefer GraphQL over URL-centric APIs such as REST?" At first glance, it seems like strictly more work and setup than traditional protocols - what do we get for the extra effort?

First, our API calls become easier to understand by **declaring the exact data we want from the server**. For a newcomer to a web app codebase, seeing GraphQL queries makes it immediately obvious what data is coming from the server versus what data is derived on the clients.

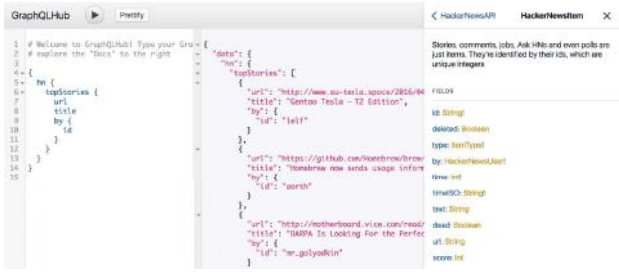
GraphQL also opens doors for better unit and integration testing: it's easy to mock data on the client, and it's possible to assert that your server GraphQL changes don't break GraphQL queries in client code.

GraphQL is also designed with performance in mind, especially with respect to mobile clients. Specifying only the data needed in each query prevents over-fetching (i.e., where the server retrieves and transmits data that ultimately goes unused by the client). This reduces network traffic and helps to improve speed on size-sensitive mobile environments.

The development experience for traditional JSON APIs is often acceptable at best (and infuriating more often). Most APIs are lacking in documentation, or worse have documentation that is inconsistent with the behavior of the API. APIs can change and it's not immediately obvious (i.e. with compile-time checks) what parts of your application will break. Very few APIs allow for discovery of properties or new endpoints, and usually it occurs in a bespoke mechanism.

GraphQL dramatically improves the developer experience - its type system provides a **living form of self-documentation**, and tooling like GraphiQL (which we play with in this chapter) allow for natural exploration of a GraphQL server.






**Navigating a Schema with GraphiQL**

Finally, the declarative nature of GraphQL pairs nicely with the declarative nature of React. A few projects, including the Relay framework from Facebook, are trying to bring the idea of "colocated queries" in React to life. Imagine writing a React component that automatically fetches the data it needs from the server, with no glue code around issuing API calls or tracking the lifecycle of a request.

All of these benefits compound as your team and product grow, which is why it evolved to be the primary way data is exchanged between client and server at Facebook.

## GraphQL vs. REST

We've mentioned alternative protocols a bit, but let's compare GraphQL to REST specifically.

One drawback to REST is "endpoint creep." Imagine you're building a social network app and start with a `/user/:id/profile` endpoint. At first the data this endpoint returns is the same for every platform and every part of the app, but slowly your product evolves and accumulates more features that fit under the "profile" umbrella. This is problematic for new parts of the app that only need *some* profile data, such as tooltips on a news feed. So you might end up creating something like `/user/:id/profile_short`, which is a restricted version of the full profile.

As you run into more cases where new endpoints are required (imagine a `profile_medium`!), you now duplicate some data with calls to `/profile` and `/profile_short`, possibly wasting server and network resources. It also makes the development experience harder to understand - where does a developer find out what data is returned in each variation? Are your docs up-to-date?

An alternative to creating *N* endpoints is to add some GraphQL-esque functionality to query parameters, such as `/user/:id/profile?include=user.name,user.id`. This allows clients to specify the data they want, but still lacks many of the features of GraphQL that make such a system work in the long-run. For example, that sort of API there is still no strong typing information that enables resilient unit testing and long-lived code. This is especially critical for mobile apps, where old binaries might exist in the wild for long periods of time.

Lastly the tooling for developing against and debugging REST APIs is often lack-luster because there are so few commonalities among popular APIs. At the lowest level you have very general purpose tools like cURL and wget, some APIs might support Swagger[95] or other documentation formats, and at the highest level for specific APIs like Elasticsearch or Facebook's Graph API you may find bespoke utilities. GraphQL's type system supports introspection (in other words, you can use GraphQL itself to discover information about a GraphQL server), which enables pluggable and portable developer tools.

## GraphQL vs. SQL

It's also worthwhile to compare GraphQL to SQL (in the abstract, not tied to a specific database or implementation). There's some precedent for using SQL for web apps with the Facebook Query Language (FQL)[96] - what makes GraphQL a better choice?

SQL is very helpful for accessing relational databases and works well with the way such databases are structured internally, but it is not how front-end applications are oriented to consume data. Dealing with concepts like joins and aggregations at the browser level feels like an abstraction leak - instead, we usually do want to think of information as a graph. We often think, "Get me this particular user, and then get me the friends of that user," where "friends" could be any type of connection, like photos or financial data.

There's also a security concern - it's awfully tempting to shove SQL from the web apps straight into the underlying databases, which will inevitably lead to security issues. And as we'll see later on, GraphQL also enables precise access control logic around who can see what kinds of data, and is generally more flexible and less likely to be as insecure as using raw SQL.

Remember that using GraphQL does not mean you have to abandon your backend SQL databases - GraphQL servers can sit on top of any data source, whether it's SQL, MongoDB, Redis, or even a third-party API. In fact, one of the benefits of GraphQL is that it is possible to write a single GraphQL sever that serves as an abstraction over several data stores (or other APIs) simultaneously.

## Relay and GraphQL Frameworks

We've talked a lot about *why* you should consider GraphQL, but haven't gone much into *how* you use GraphQL. We'll start to uncover more about that very soon, but we should mention Relay.

Relay[97] is Facebook's framework for connecting React components to a GraphQL server. It allows you to write code like this, which shows how an `Item` component can automatically retrieve data from a Hacker News GraphQL server:

[95] http://swagger.io/
[96] https://en.wikipedia.org/wiki/Facebook_Query_Language
[97] https://facebook.github.io/relay/



```
class Item extends React.Component {
  render() {
    let item = this.props.item;

    return (
      <div>
        <h1><a href={item.url}>{item.title}</a></h1>
        <h2>{item.score} - {item.by.id}</h2>
        <hr />
      </div>
    );
  }
};

Item = Relay.createContainer(Item, {
  fragments: {
    item: () => Relay.QL`
      fragment on HackerNewsItem {
        id
        title,
        score,
        url
        by {
          id
        }
      }
    `
  },
});
```

Behind the scenes, Relay handles intelligently batching and caching data for improved performance and a consistent user experience. We'll go in-depth on Relay later on, but this should give you an idea of how nicely GraphQL and React can work together.

There are other emerging approaches on how to integrate GraphQL and React, such as Apollo[98] by the Meteor team.

**You can also use GraphQL without using React** - it's easy to use GraphQL anywhere you'd traditionally use API calls, including with other technologies like Angular or Backbone.

[98] http://www.apollostack.com/

## 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 severs, 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[99]. 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.

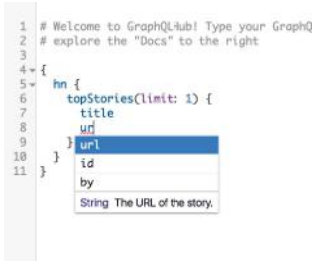
Head to https://graphqlhub.com/playground[100] and get your first look at GraphiQL:

[99] https://github.com/graphql/graphiql
[100] https://graphqlhub.com/playground



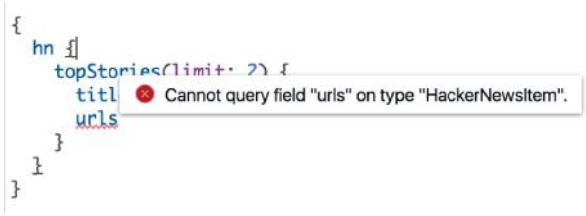

**Empty GraphiQL**

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**

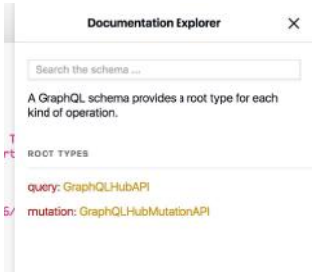
This is a great example of GraphQL's type system at work. GraphiQL knows what fields and types

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:



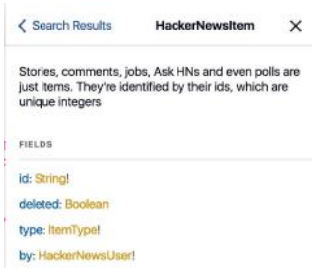

GraphiQL docs

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[101].

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:

[101] https://facebook.github.io/graphql/



```
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 optimizations[102] detailed by Facebook.

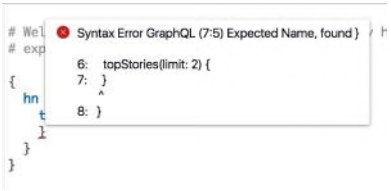
So that's documents and operations, now let's dig into a typical query. An operation is composed of *selections*, which are generally *fields*. Each field in GraphQL represents a piece of data, which can either be an irreducible *scalar* type (defined below) or a more complex type composed of yet more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

[102] https://github.com/facebook/graphql/issues/29#issuecomment-118994619

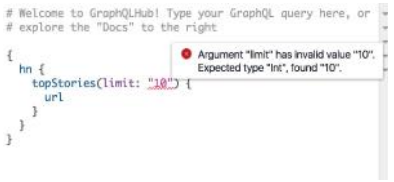


Scalar types include `Int`, `Float`, `String`, `Boolean` and `ID` (coerced to a string). GraphQL provides ways of composing these scalars into more complex types using `Object`, `Interface`, `Union`, `Enum`, and `List` types. We'll go into each of those later, but it should be intuitive that they allow 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 not just the string or numeric scalars we've shown, but are arbitrarily deeply nested maps of keys

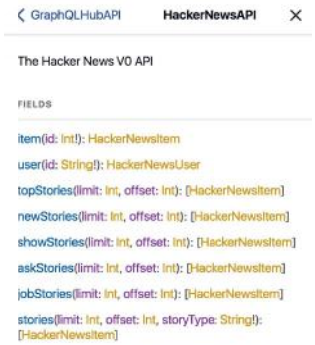
and values. Here's an example where the argument `storyData` takes an input object which has a `url` property:

```
{
  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[103]:

```
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:

[103] https://facebook.github.io/graphql/#sec-Unions

```
{
  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

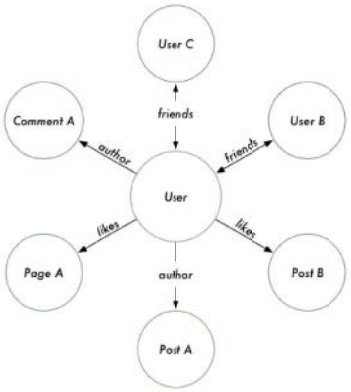
We've explored the "QL" of GraphQL, but haven't touched too much on the "Graph" part.

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 sense[104].

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:

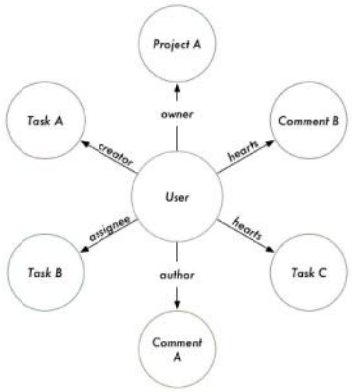
[104] https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)




**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
        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 have provides 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[105]. 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. You 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:

[105] https://facebook.github.io/relay/docs/graphql-relay-specification.html



```
{
  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[106]'s Hacker News account). We then grab the first two nodes in the `submitted` connection.

[106] https://en.wikipedia.org/wiki/Drew_Houston

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 real-time 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[107]:

```
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[108] with the `clientSubscriptionId`, but other possibilities include

[107] http://graphql.org/blog/subscriptions-in-graphql-and-relay/
[108] https://en.wikipedia.org/wiki/MQTT

WebSockets[109], Server-Sent Events[110], 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[111], 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 website[112] 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:

[109] https://en.wikipedia.org/wiki/WebSocket
[110] https://en.wikipedia.org/wiki/Server-sent_events
[111] http://graphql.org/blog/subscriptions-in-graphql-and-relay/#why-not-live-queries
[112] https://www.graphqlhub.com/



```
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[113] is a collection of projects including `react-apollo`[114]. `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:

[113] http://apollostack.com
[114] http://docs.apollostack.com/apollo-client/react.html



```
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())
);
```

Check out the Apollo docs[115] if this sounds like it could be a good fit for your project.

## Wrapping Up

Now you've written a few GraphQL queries, learned about different its features, and even written a bit of code to get GraphQL in your browser. If you have an existing GraphQL server for your product, it might be okay to move on and jump to the chapter on Relay - but in most cases you'll also need to draft your own GraphQL server. Excelsior!

[115] http://docs.apollostack.com/index.html

# GraphQL Server

## Writing a GraphQL Server

In order to use Relay or any other GraphQL library, you need a server that speaks GraphQL. In this chapter, we're going to write a backend GraphQL server with NodeJS and other technologies we've used in earlier chapters.

We're using Node because we can leverage the tooling used elsewhere in the React ecosystem, and Facebook targets Node for many of their backend libraries. However, there are GraphQL server libraries in every popular language, such as Ruby[116], Python[117], and Scala[118]. If your existing backend uses a framework in a language other than JavaScript, such as Rails, it might make sense to look into a GraphQL implementation in your current language.

The lessons we'll go over in this section, such as how to design a schema and work with an existing SQL database, are applicable to all GraphQL libraries and languages. We encourage you to follow along with the section and apply what you learn to your own projects, regardless of language.

Let's get to it!

## Special setup for Windows users

Windows users require a little additional setup to install the packages for this chapter. Specifically, the `sqlite3` package that we use can be a bit difficult to install on some Windows versions.

**1. Install `windows-build-tools`**

`windows-build-tools` allows you to compile native Node modules, which is necessary for the `sqlite3` package.

After you've setup Node and npm, install `windows-build-tools` globally:

```
npm install --global --production windows-build-tools
```

**2. Add `python` to your PATH**

After installing `windows-build-tools`, you need to ensure `python` is in your PATH. This means that typing `python` into your terminal and pressing enter should invoke Python.

`windows-build-tools` installs Python here:

[116] https://github.com/rmosolgo/graphql-ruby
[117] https://github.com/graphql-python/graphene
[118] https://github.com/sangria-graphql/sangria



```
C:\<Your User>\.windows-build-tools\python27
```

Python comes with a script to add itself to your PATH. To run that script in PowerShell:

```
> $env:UserProfile\.windows-build-tools\python27\scripts\win_add2path.py
```

If you're getting an error that this script is not found, verify the version number for Python above is correct by looking inside the `C:\<Your User>\.windows-build-tools` directory.

After running this, **you must restart your computer**. Sometimes just restarting PowerShell works. In any case, you can verify that Python is properly installed by invoking `python` in the terminal. Doing so should start a Python console:

```
> python
```

## Game Plan

At a high-level, here's what we're going to do:

- Create an Express[119] HTTP server
- Add an endpoint which accepts GraphQL requests
- Construct our GraphQL schema
- Write the glue-code that resolves data for each GraphQL field in our schema
- Support GraphiQL so we can debug and iterate quickly

The schema we're going to draft is going to be for a social network, a sort of "Facebook-lite," backed by a SQLite database. This will show common GraphQL patterns and techniques to efficiently engineer GraphQL servers talking to existing data stores.

## Express HTTP Server

Let's start setting up our web server. Create a new directory called `graphql-server` and run some initial `npm` commands:

[119] http://expressjs.com

```
$ mkdir graphql-server
$ cd ./graphql-server
$ npm init
# hit enter a bunch, accept the defaults
$ npm install babel-register@6.3.13 babel-preset-es2015@6.3.13 express@4.13.3 --\
save --save-exact
$ echo '{ "presets": ["es2015"] }' > .babelrc
```

Let's run through what happened: we created a new folder called `graphql-server` and then jumped inside of it. We ran `npm init`, which creates a `package.json` for us. Then we installed some dependencies, Babel and Express. The name Babel should be familiar from earlier chapters - in this case, we installed `babel-register` to transpile NodeJS files and `babel-preset-es2015` to instruct Babel on how to transpile said files. The final command created a file called `.babelrc`, which configured Babel to use the `babel-preset-es2015` package.

Create a new file named `index.js`, open it, and add these lines:

```
require('babel-register');

require('./server');
```

Not a lot going on here, but it's important. By requiring `babel-register`, every subsequent call to `require` (or `import` when using ES2015) will go through Babel's transpiler. Babel will transpile the files according to the settings in `.babelrc`, which we configured to use the `es2015` settings.

For our next trick, create a new file named `server.js`. Open it and add a quick line to debug that our code is working:

```
console.log({ starting: true });
```

If you run `node index.js`, you should see this happen:

```
$ node index.js
{ starting: true }
```

Wonderful start! Now let's add some HTTP.

Express is a very powerful and extensible HTTP framework, so we're not going to go too in-depth; if you're ever curious to learn more about it, check out their documentation[120].

Open up `server.js` again and add code to configure Express:

[120] http://expressjs.com



```
console.log({ starting: true });

import express from 'express';

const app = express();

app.use('/graphql', (req, res) => {
  res.send({ data: true });
});

app.listen(3000, () => {
  console.log({ running: true });
});
```

The first few lines are straight-forward - we import the `express` package and create a new instance (you can think of this as creating a new server). At the end of the file, we tell that server to start listening for traffic on port 3000 and show some output after that's happening.

But before we start the server, we need to tell it how to handle different kinds of requests. `app.use` is how we're going to do that today. It's first argument is the path to handle, and the second argument is a handler function. `req` and `res` are shorthand for "request" and "response", respectively. By default, paths registered with `app.use` will respond on all HTTP methods, so as of now `GET /graphql` and `POST /graphql` do the same thing.

Let's give a shot and test it out. Run your server again with `node index.js`, and in a separate terminal fire off a cURL:

```
$ node index.js
{ starting: true }
{ running: true }
```

```
$ curl -XPOST http://localhost:3000/graphql
{"data":true}
$ curl -XGET http://localhost:3000/graphql
{"date":true}
```

We have a working HTTP server! Now time to "do some GraphQL," so to speak.

Tired of restarting your server after every change? You can setup a tool like Nodemon[121] to automatically restart your server when you make edits. `npm install -g nodemon && nodemon index.js` should do the trick.

[121] https://github.com/remy/nodemon#nodemon

## Adding First GraphQL Types

We need to install some GraphQL libraries; stop your server if it's running, and run these commands:

```
$ npm install graphql@0.6.0 express-graphql@0.5.3 --save --save-exact
```

Both have "GraphQL" in their name, so that should sound promising. These are two libraries maintained by Facebook and also serve as reference implementations for GraphQL libraries in other languages.

The `graphql` library[122] exposes APIs that let us construct our schema, and then exposes an API for resolving raw GraphQL document strings against that schema. It can be used in any JavaScript application, whether an Express web server like in this example, or another servers like Koa, or even in the browser itself.

In contrast, the `express-graphql` package[123] is meant to be used only with Express. It handles ensuring that HTTP requests and responses are correctly formatted for GraphQL (such dealing with the `content-type` header), and will eventually allows us to support GraphiQL with very little extra work.

Let's get to it - open up `server.js` and add these lines after you create the `app` instance:

```
5  const app = express();
6
7  import graphqlHTTP from 'express-graphql';
8  import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';
9
10 const RootQuery = new GraphQLObjectType({
11   name: 'RootQuery',
12   description: 'The root query',
13   fields: {
14     viewer: {
15       type: GraphQLString,
16       resolve() {
17         return 'viewer!';
18       }
19     }
20   }
21 });
22
23 const Schema = new GraphQLSchema({
```

[122] https://github.com/graphql/graphql-js
[123] https://github.com/graphql/express-graphql



```
24   query: RootQuery
25 });
26
27 app.use('/graphql', graphqlHTTP({ schema: Schema }));
```

Note that we've changed our previous arguments to `app.use` (this replaces the `app.use` from before).

There's a bunch of interesting things going on here, but let's skip to the good part first. Start up your server (`node index.js`) and run this cURL command:

```
$ curl -XPOST -H 'content-type:application/graphql' http://localhost:3000/graphq\
l -d '{ viewer }'
{"data":{"viewer":"viewer!"}}
```

If you see the above output then your server is configured correctly and resolving GraphQL requests accordingly. Now let's walk through how it actually works.

First we import some dependencies from the GraphQL libraries:

```
7  import graphqlHTTP from 'express-graphql';
8  import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';
```

The `graphql` library exports many objects and you'll become familiar with them as we write more code. The first two we use are `GraphQLObjectType` and `GraphQLString`:

```
10 const RootQuery = new GraphQLObjectType({
11   name: 'RootQuery',
12   description: 'The root query',
13   fields: {
14     viewer: {
15       type: GraphQLString,
16       resolve() {
17         return 'viewer!';
18       }
19     }
20   }
21 });
```

When you create a new instance of `GraphQLObjectType`, it's analogous to defining a new class. It's required that we give it a `name` and optional (but very helpful for documentation) that we set a `description`.

The `name` field sets the type name in the GraphQL schema. For instance, if want to define a fragment on this type, we would write `... on RootQuery` in our query. If we changed `name` to something like `AwesomeRootQuery`, we would need to change our fragment to `... on AwesomeRootQuery`, even though the JavaScript variable is still `RootQuery`.

That defines the type, now we need to give it some fields. Each key in the `fields` object defines a new corresponding field, and each field object has some required properties.

We need to give it:

- a `type` - the GraphQL library exports the basic scalar types, such as `GraphQLString`.
- a `resolve` function to return the **value** of the field - for now, we have the hard-coded value `'viewer!'`.

Next we create an instance of `GraphQLSchema`:

```
23 const Schema = new GraphQLSchema({
24   query: RootQuery
25 });
```

Hopefully the naming makes it clear that this is the top-level GraphQL object.

You can only resolve queries once you have an instance of a schema - you can't resolve query strings against object types by themselves.

Schemas have two properties: `query` and `mutation`, which corresponds to the two types of operations we discussed earlier. Both of these take an instance of a GraphQL type, and for now we just set the `query` to `RootQuery`.

One quick note on naming things (one of the Hard Problems of computer science): we generally refer to the top-level query of a schema as the *root*. You'll see many projects that have similar `RootQuery`-named types.

Finally we hook it all up to Express:

```
27 app.use('/graphql', graphqlHTTP({ schema: Schema }));
```

Instead of manually writing a handler function, the `graphqlHTTP` function will generate one using our `Schema`. Internally, this will grab our GraphQL query from the request and hand it off to the main GraphQL library's resolving function.

## Adding GraphiQL

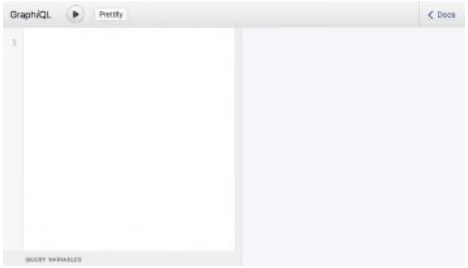
Earlier we used GraphQLHub's hosted instance of GraphiQL, the GraphQL IDE. What if I told you that you could add GraphiQL to our little GraphQL server with just one change?

Try adding the `graphiql: true` option to `graphqlHTTP`:



```
27 app.use('/graphql', graphqlHTTP({ schema: Schema, graphiql: true }));
```

Restart your server, head to Chrome, and open `http://localhost:3000/graphql`. You should see something like this:

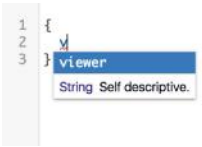



**Empty GraphiQL**

If you open the "Docs" sidebar, you'll see all the information we entered about our Schema - the `RootQuery`, the description, and it's `viewer` field:


**Server Docs**

You'll also get auto-complete for our fields:




**Server Autocomplete**

We get all of this goodness for free by using `graphql-express`.

It's also possible to setup GraphiQL if you're using another JavaScript framework or an entirely different language, read GraphiQL's documentation[124] for details.

You'll notice that our typeahead suggests some interesting fields like `__schema`, even though we didn't define that. This is something we've alluded to throughout the chapter: GraphQL's introspection features.


**Server Introspection**

Let's dig into it a bit further.

## Introspection

Go ahead and run this GraphQL query inside of our server's GraphiQL instance:

```
{
  __schema {
    queryType {
      name
      fields {
        name
        type {
          name
        }
      }
    }
  }
}
```

`__schema` is a "meta" field that automatically exists on every GraphQL root query operation. It has a whole tree of its own types and their fields, which you can read about in the GraphQL introspection spec[125]. GraphiQL's auto-complete will also give you helpful information and let you explore the possibilities of `__schema` and the other introspection field, `__type`.

After running that query, you'll get some data back like this:

[124] https://github.com/graphql/graphiql#getting-started
[125] https://facebook.github.io/graphql/#sec-Schema-Introspection



```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "RootQuery",
        "fields": [
          {
            "name": "viewer",
            "type": {
              "name": "String"
            }
          }
        ]
      }
    }
  }
}
```

This is essentially a JSON description of our server's schema. This is how GraphiQL populates it's documentation and auto-complete, by issuing an introspection query as the IDE boots up. Since every GraphQL server is required to support introspection, tooling is often portable across all GraphQL servers (again, regardless of the language or library they were implemented with).

We won't do anything else with introspection for this chapter, but it's good to know that it exists and how it works.

## Mutation

So far we've set the root `query` of our schema, but we mentioned that we can also set the root `mutation`. Remember from earlier that mutations are the correct place for "writes" to happen in GraphQL - whenever you want to add, update, or remove data, it should occur through a mutation operation. As we'll see, mutations differ very little from queries other than the implicit contract that writes should **not** happen in queries.

To demonstrate mutations, we'll create a simple API to get and set in-memory `node` objects, similar to the idiomatic `Node` pattern we described earlier. For now, instead of returning objects that implement a `Node` interface, we're going to return a `string` for our nodes.

Let's start by importing some new dependencies from the GraphQL library:

```
import { GraphQLSchema, GraphQLObjectType, GraphQLString,
  GraphQLNonNull, GraphQLID } from 'graphql';
```

`GraphQLID` is the JavaScript analog to the `ID` scalar type, while `GraphQLNonNull` is something we haven't quite covered yet. It turns out that GraphQL's type system not only tracks the types and interfaces of fields, but also whether or not they can be null. This is especially handy for field arguments, which we'll get to in a second.

Next we need to create a type for our mutation. Take a second to think about what that code will look like, given that it should be fairly similar to our `RootQuery` type. What kinds of invocations and properties will we need?

After you've given it some thought, compare it to the actual implementation:

```
35 let inMemoryStore = {};
36 const RootMutation = new GraphQLObjectType({
37   name: 'RootMutation',
38   description: 'The root mutation',
39   fields: {
40     setNode: {
41       type: GraphQLString,
42       args: {
43         id: {
44           type: new GraphQLNonNull(GraphQLID)
45         },
46         value: {
47           type: new GraphQLNonNull(GraphQLString),
48         }
49       },
50       resolve(source, args) {
51         inMemoryStore[args.key] = args.value;
52         return inMemoryStore[args.key];
53       }
54     }
55   }
56 });
```

At the very top we initialize a "store" for our nodes. This will only live in-memory, so whenever you restart the server it will clear the data - but you can start to imagine this being a key-value service like Redis or Memcached. Creating an instance of `GraphQLObjectType`, setting the `name`, `description`, and `fields` should all look familiar.

One new thing here is dealing with field arguments using the `args` property. Similar to how we set the names of fields with the `fields` property, the keys of the `args` object are the names of the arguments allowed by the field.

We specify each argument's type, wrapped as an instance of `GraphQLNonNull`. For arguments, this means that the query **must** specify a non-null value for each argument. Our `resolve` function



takes this into account - `resolve` gets passed several arguments, the second of which is an object containing the field arguments (we'll touch on `source` later on).

When we set a value in `inMemoryStore`, that is the "write" of our mutation. We also return a value in the `resolve` function, to cooperate with the `type` of the `setNode` field. In this case it happens to be a `string`, but you can imagine it being a complex type like `User` or something bespoke for the mutation.

Now that we have our mutation type, we add it to our schema:

```
58 const Schema = new GraphQLSchema({
59   query: RootQuery,
60   mutation: RootMutation,
61 });
```

For one last bit of housekeeping, we'll go ahead and add a `node` field to our query:

```
14   fields: {
15     viewer: {
16       type: GraphQLString,
17       resolve() {
18         return 'viewer!';
19       }
20     },
21     node: {
22       type: GraphQLString,
23       args: {
24         id: {
25           type: new GraphQLNonNull(GraphQLID)
26         }
27       },
28       resolve(source, args) {
29         return inMemoryStore[args.key];
30       }
31     }
32   }
```

Restart your server and give this query a run in GraphiQL:

```
mutation {
  setNode(id: "id", value: "a value!")
}
```

Notice that we have to explicitly state the `mutation` operation type, since GraphQL assumes `query` otherwise. Running the mutation should return the new value:

```
{
  "data": {
    "node": "a value!"
  }
}
```

You can then confirm that the mutation worked by running a fresh query:

```
query {
  node(id: "id")
}
```

Mutations aren't too conceptually complicated, and most apps will need them to write data back to the server eventually. Relay has a slightly more rigorous pattern to defining mutations, which we'll get to in due time.

This mutation changed an in-memory data structure, but most products' data lives in datastores like Postgres or MySQL. It's time to explore how we work with those environments.

## Rich Schemas and SQL

As we mentioned earlier, we're going to build a small Facebook clone using SQLite. It's going to show how to communicate with a database, how to handle authorization and permissions, and some performance tips to make it runs smoothly.

Before we dive into the code, here's what our database is going to look like:

- A `users` table, which has `id`, `name`, and `about` columns
- A `users_friends` table, which has `user_id_a`, `user_id_b`, and `level` columns
- A `posts` column, which has `body`, `user_id`, `level`, and `created_at` columns

The `level` columns are going to represent a hierarchical "privacy" setting for friendships and posts. The possible levels we'll use are `top`, `friend`, `acquaintance`, and `public`. If a post has a `level` of `acquaintance`, then only friends with that level or "higher" can see it. `public` posts can be seen by anyone, even if the user isn't a friend.



To implement this in NodeJS, we're going to use the `node-sql`[126] and `sqlite3`[127] packages. There are many options when working with database in NodeJS, and you should do your own research before using any mission-critical libraries in production, but these should suffice for our learning.

## Setting Up The Database

Time to install some more libraries! Run this to add them to our project:

```
$ npm install sqlite@0.0.4 sqlite3@3.1.3 --save --save-exact
$ mkdir src
```

We'll eventually want three files. On macOS or Linux, you can run:

```
$ touch src/tables.js src/database.js src/seedData.js
```

Windows users can create them as we go.

Now let's create a database. SQLite databases exist in files, so there's no need to start a separate process or install more dependencies. For legibility, we're going to start splitting our code into multiple files, which is the set of files we created with `touch`.

First we define our tables - you can read `node-sql`'s documentation for specifics, but it's pretty legible. Open up `tables.js` and add these definitions:

```
import sql from 'sql';

sql.setDialect('sqlite');

export const users = sql.define({
  name: 'users',
  columns: [{
    name: 'id',
    dataType: 'INTEGER',
    primaryKey: true
  }, {
    name: 'name',
    dataType: 'text'
  }, {
    name: 'about',
    dataType: 'text'
```

[126] https://github.com/brianc/node-sql
[127] https://github.com/mapbox/node-sqlite3

```
  }]
});

export const usersFriends = sql.define({
  name: 'users_friends',
  columns: [{
    name: 'user_id_a',
    dataType: 'int',
  }, {
    name: 'user_id_b',
    dataType: 'int',
  }, {
    name: 'level',
    dataType: 'text',
  }]
});

export const posts = sql.define({
  name: 'posts',
  columns: [{
    name: 'id',
    dataType: 'INTEGER',
    primaryKey: true
  }, {
    name: 'user_id',
    dataType: 'int'
  }, {
    name: 'body',
    dataType: 'text'
  }, {
    name: 'level',
    dataType: 'text'
  }, {
    name: 'created_at',
    dataType: 'datetime'
  }]
});
```

`node-sql` lets us craft and manipulate SQL queries using JavaScript objects, similar to how the GraphQL JavaScript library enables us to work with GraphQL. There's nothing "happening" in this particular file, but we'll consume the objects it exports soon.

Now we need to create the database and load it with some data. Included with the materials for this



course, inside the `graphql-server/src` directory, is a `data.json` file. Go ahead and copy that into the `src` directory of the project we're working on. You can also come up with your own data, but the examples we'll work through will assume you're using the included data file.

To copy the data from `data.json` into the database, we need to write a bit of code. First open up `src/database.js` and define some simple exports:

```
import sqlite3 from 'sqlite3';

import * as tables from './tables';

export const db = new sqlite3.Database('./db.sqlite');

export const getSql = (query) => {
  return new Promise((resolve, reject) => {
    console.log(query.text);
    console.log(query.values);
    db.all(query.text, query.values, (err, rows) => {
      if (err) {
        reject(err);
      } else {
        resolve(rows);
      }
    });
  });
```

First we define our database file and export it so other code can use it. We also export a `getSql` function, which will run our queries as asynchronous promises. The GraphQL JS library uses promises to handle asynchronous resolving, which we'll leverage soon.

Then open up the `src/seedData.js` file we created earlier and start by adding this `createDatabase` function:

```
import * as data from './data';
import * as tables from './tables';
import * as database from './database';

const sequencePromises = function (promises) {
  return promises.reduce((promise, promiseFunction) => {
    return promise.then(() => {
      return promiseFunction();
    });
  }, Promise.resolve());
```

```
};

const createDatabase = () => {
  let promises = [tables.users, tables.usersFriends, tables.posts].map((table) =\
> {
    return () => database.getSql(table.create().toQuery());
  });

  return sequencePromises(promises);
};
```

Recall that the exports of `tables.js` are the objects created by `node-sql`. When we want to create a database query with `node-sql`, we use the `.toQuery()` function and then pass it into the `getSql` function we just wrote in `database.js`. In plain-English, our new `createDatabase` function runs queries that create each table. We use `Promise.all` to make sure all of the tables are created before moving on to any next steps in the promise chain.

After the tables are setup, we need to add our data from `data.json`. Write this new `insertData` function next:

```
const insertData = () => {
  let { users, posts, usersFriends } = data;

  let queries = [
    tables.users.insert(users).toQuery(),
    tables.posts.insert(posts).toQuery(),
    tables.usersFriends.insert(usersFriends).toQuery(),
  ];

  let promises = queries.map((query) => {
    return () => database.getSql(query);
  });
  return sequencePromises(promises);
};
```

Similar to `createDatabase`, we create queries using `toQuery` and then execute them using `getSql`. Finally we tie it all together at the end of the file by invoking our two functions:



```
createDatabase().then(() => {
  return insertData();
}).then(() => {
  console.log({ done: true });
});
```

To get this code to run, you can invoke this shell command:

```
$ node -e 'require("babel-register"); require("./src/seedData");'
{ done: true }
```

You should now have a `db.sqlite` file at the top of your project! You can use many graphical tools to explore your SQLite database, such as DBeaver[128], or you can verify that it has some data with this one-line command:

```
$ sqlite3 ./db.sqlite "select count(*) from users"
5
```

Now it's time to hook up the GraphQL schema to our newly created database.

## Schema Design

In our earlier examples, the `resolve` functions in our GraphQL schema would just return constant or in-memory values, but now they need to return data from our database. We also need corresponding GraphQL types for our `users` and `posts` table, and to add the corresponding fields to our original root query.

Before we dive into the code, we should take a minute to consider how our final GraphQL queries are going to look. Generally it's a good practice to start with the `viewer`, since most of our data will flow through that field.

In this case, the `viewer` will be the current user. A user has `friends`, so we'll need a list for that, as well as a connection for the `posts` authored by that user. The `viewer` also has a `newsfeed`, which is a connection to posts authored by other users. We'll want all of these connections to be idiomatic GraphQL and include features like proper pagination, in case the entire newsfeed or friends list gets too long to compute or send in one response.

Given all of that information, we expect our `viewer` to enable queries like this:

[128] http://dbeaver.jkiss.org/

```
{
  viewer {
    friends {
      # connection fields for Users
    }

    posts {
      # connection fields for Posts
    }

    newsfeed {
      # connection fields for Posts
    }
  }
}
```

Remember that we want all of our objects to also be nodes in a graph, in accordance with idiomatic GraphQL. Eventually, all of the data returned in `newsfeed` will need to be authorization-aware, taking into account the friendship `level` between the author of a given post and the `viewer`.

We should add support for queries that fetch arbitrary nodes using a top-level `node(id:)` field like this:

```
{
  node(id: "123") {
    ... on User {
      friends {
        # friends
      }
    }

    ... on Post {
      author {
        posts {
          # connection fields
        }
      }

      body
    }
  }
}
```



As we mentioned in an earlier section, this field is valuable to help front-end code re-fetch the current state of any node without knowing it's position in the hierarchy.

This may be surprising, but the GraphQL convention is to fetch lots of different types of objects from the same top-level `node` field. SQL databases usually have a different table for each type, and REST APIs use distinct endpoints per type, but idiomatic GraphQL fetches all objects the same way as long as you have an identifier. This also means that IDs need to be globally unique, or else a GraphQL server can't tell the difference between a `User` with `id: 1` and a `Post` with `id: 1`.

## Object and Scalar Types

To start making these queries possible, we'll create a new file to hold these types called `src/types.js`.

The first type we need to define is the `Node` interface. Recall that for a type to be a valid `Node`, it needs to have a globally-unique `id` field. To implement that in JavaScript, we start by importing some APIs and creating an instance of `GraphQLInterfaceType`:

```
import {
  GraphQLInterfaceType,
  GraphQLObjectType,
  GraphQLID,
  GraphQLString,
  GraphQLNonNull,
  GraphQLList,
} from 'graphql';

import * as tables from './tables';

export const NodeInterface = new GraphQLInterfaceType({
  name: 'Node',
  fields: {
    id: {
      type: new GraphQLNonNull(GraphQLID)
    }
```

Aside from using a new class, everything looks familiar to how we create instance of `GraphQLObjectType`. Notice that the `id` field does *not* have a `resolve` function. Fields in interfaces are not expected to have default implementations of `resolve`, and even if you do provide one it will be ignored. Instead, each object type that implements `Node` should define its own `resolve` (we'll get to that in a second).

In addition to defining `fields`, `GraphQLInterfaceType` instances must also define a `resolveType` function. Remember that our top-level `node(id:)` field only guarantees that some kind of `Node` will

be returned, it does not make any guarantees about which specific type. In order for GraphQL to do further resolution (such as using partial fragments, i.e. `... on User`), we need to inform the GraphQL engine of the concrete type for a particular object.

We can implement it like this:

```
export const NodeInterface = new GraphQLInterfaceType({
  name: 'Node',
  fields: {
    id: {
      type: new GraphQLNonNull(GraphQLID)
    }
  },
  resolveType: (source) => {
    if (source.__tableName === tables.users.getName()) {
      return UserType;
    }
    return PostType;
  }
});
```

The `resolveType` function takes in raw data as its first argument (in this case, `source` will be the data returned directly from the database), and is expected to return an instance of `GraphQLObjectType` that implement the interface. We use the `__tableName` property of `source`, which isn't an actual column in the database - we'll see how that gets injected later.

We return some objects, `UserType` and `PostType`, that we haven't defined quite yet. Add their definitions below `resolveType`:

```
const resolveId = (source) => {
  return tables.dbIdToNodeId(source.id, source.__tableName);
};

export const UserType = new GraphQLObjectType({
  name: 'User',
  interfaces: [ NodeInterface ],
  fields: {
    id: {
      type: new GraphQLNonNull(GraphQLID),
      resolve: resolveId
    },
    name: {
      type: new GraphQLNonNull(GraphQLString)
    },
    about: {
      type: new GraphQLNonNull(GraphQLString)
    }
  }
});

export const PostType = new GraphQLObjectType({
  name: 'Post',
  interfaces: [ NodeInterface ],
  fields: {
    id: {
      type: new GraphQLNonNull(GraphQLID),
      resolve: resolveId
    },
    createdAt: {
      type: new GraphQLNonNull(GraphQLString),
    },
    body: {
      type: new GraphQLNonNull(GraphQLString)
    }
  }
```



Most of the code should look similar to everything we've been working with so far. We define new instances of `GraphQLObjectType`, add `NodeInterface` to their `interfaces` property, and implement their `fields`. Some of the `fields` are wrapped in `GraphQLNonNull`, which enforces that they must exist. If you don't provide an implementation of `resolve` to a field, it will do a simple property lookup on the underlying `source` data - for example, the `name` field will invoke `source.name`.

We do provide an implementation of `resolve` for `id` on both of these types, which both use the same `resolveId` function. Even though our `NodeInterface` code couldn't provide a default `resolve`, we can still share code by referencing the same variable.

Our implementation of `resolveId` uses `tables.dbIdToNodeId`, which we haven't defined in `tables.js`. Open up `tables.js` and add these two new exported functions at the bottom:

```
export const dbIdToNodeId = (dbId, tableName) => {
  return `${tableName}:${dbId}`;
};

export const splitNodeId = (nodeId) => {
  const [tableName, dbId] = nodeId.split(':');
  return { tableName, dbId };
};
```

Earlier we mentioned that `Node` IDs must be globally unique, but looking at our `data.json` we have some row ID collisions. This is not uncommon in relational databases like Postgres and MySQL, so we have to write some logic which coerces the row integer ID into a unique string. In a production application, you may want to obfuscate IDs to leak less information about your database, but using the raw table name will make it easier to debug for now.

We're almost ready to run a GraphQL query! Head to `server.js`, import some of the types we just authored, and change the `RootQuery` to have only the new `node` field that we want:

```
import {
  NodeInterface,
  UserType,
  PostType
} from './src/types';

import * as loaders from './src/loaders';

const RootQuery = new GraphQLObjectType({
  name: 'RootQuery',
  description: 'The root query',
  fields: {
    node: {
      type: NodeInterface,
      args: {
        id: {
          type: new GraphQLNonNull(GraphQLID)
        }
      },
      resolve(source, args) {
        return loaders.getNodeById(args.id);
      }
    }
  }
});
```



We also imported a new file, `src/loaders.js`, which we need to edit. Its purpose will be to expose APIs that load data from the source - we don't want to clutter our server or top-level schema code with code that directly accesses the database.

Create that `src/loaders.js` file and add this small function:

```
import * as database from './database';
import * as tables from './tables';

export const getNodeById = (nodeId) => {
  const { tableName, dbId } = tables.splitNodeId(nodeId);

  const table = tables[tableName];
  const query = table
    .select(table.star())
    .where(table.id.equals(dbId))
    .limit(1)
    .toQuery();

  return database.getSql(query).then((rows) => {
    if (rows[0]) {
      rows[0].__tableName = tableName;
    }
    return rows[0];
  });
};
```

This should look similar to the code we wrote in `seedData` - we use the `node-sql` APIs to construct a SQL query based on the `nodeId` provided. Remember that this `nodeId` is the globally-unique node ID, not a row ID, so we extract the database-specific information with `tables.splitNodeId`.

One trick we perform is attaching the `__tableName` property. This helps us in our earlier `resolveType` function, and anywhere else we may need to tie an object back to its underlying table. It's safe to add this because we don't expose the `__tableName` property explicitly in the GraphQL schema, so malicious consumers can't access it.

A very subtle but important thing happening with all of this code is that `loaders.getNodeById` returns a `Promise` (which is why we can attach the `__tableName` using `.then`), and ultimately that promise is returned in `resolve`. Promises are how GraphQL handles asynchronous field resolution, which usually occurs with database queries and any third-party API calls. If you're using an API that doesn't natively support promises, you can refer to the Promise API[129] to convert callback-based APIs to promises.

[129] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#Creating_a_Promise

One last step, we need to tell our `GraphQLSchema` about all the possible types in our schema. You have to do this if you use interfaces, so GraphQL can compute the list of types that implement any interfaces.

```
const Schema = new GraphQLSchema({
  types: [UserType, PostType],
  query: RootQuery,
  mutation: RootMutation,
});
```

And that's it! Restart your server, open up GraphiQL (still available at `http://localhost:3000/graphql`), and try this query:

```
{
  node(id:"users:4") {
    id
    ... on User {
      name
    }
  }
}
```

You should see this data returned:

```
{
  "data": {
    "node": {
      "id": "users:4",
      "name": "Roger"
    }
  }
}
```

You can play around with other node IDs, such as `"posts:4"` and `... on Post`. Before we move on, reflect on what's going on under the hood: we request a particular node ID, which invokes a database call, and then each field gets `resolve`'d against the database `source` data.

We could write a very simple front-end app against our current server at this point, but we have more of our schema to fill-out. Let's work on some of those `friends` list and `posts` connections.

## Lists

We mentioned earlier that the `friends` field should return a list of `User` types. Let's take baby steps towards that goal and return a simple list of IDs for now.

First let's edit our `UserType`. Open up `types.js` and a new `friends` field:



```
    about: {
      type: new GraphQLNonNull(GraphQLString)
    },
    friends: {
      type: new GraphQLList(GraphQLID),
      resolve(source) {
        return loaders.getFriendIdsForUser(source).then((rows) => {
          return rows.map((row) => {
            return tables.dbIdToNodeId(row.user_id_b, row.__tableName);
          });
        })
      }
    }
```

We set the `friends` field to return a `GraphQLList` of IDs. `GraphQLList` works similarly to the `GraphQLNonNull` we saw earlier, wrapping an inner type in its constructor. Inside `resolve` we invoke a new loader (to-be-written), and then coerce its results to the globally-unique IDs we expect.

Add an `import` at the top of the file to prepare for our new loader:

```
import * as tables from './tables';
import * as loaders from './loaders';
```

Edit `loaders.js` with that new `getFriendIdsForUser` function:

```
export const getFriendIdsForUser = (userSource) => {
  const table = tables.usersFriends;
  const query = table
    .select(table.user_id_b)
    .where(table.user_id_a.equals(userSource.id))
    .toQuery();

  return database.getSql(query).then((rows) => {
    rows.forEach((row) => {
      row.__tableName = tables.users.getName();
    });
    return rows;
  });
};
```

That's all the new code we have to write - fire up GraphiQL and run this query:

```
{
  node(id:"users:4") {
    id
    ... on User {
      name
      friends
    }
  }
}
```

Confirm that your results are equal to this:

```
{
  "data": {
    "node": {
      "id": "users:4",
      "name": "Roger",
      "friends": [
        "users:1",
        "users:3",
        "users:2"
      ]
    }
  }
}
```

## Performance: Look-Ahead Optimizations

This is great, but there's some sneaky business going on under-the-hood that we should explore. When we resolve the original `node` field, that executes one database query (`loaders.getNodeById`). Then after that node is resolved, we execute *another* database query (`loaders.getFriendIdsForUser`). If you extend this pattern to a larger application, you can imagine lots of database queries getting fired - at worst, one per field. But we know that in SQL it's possible to express these two queries with one efficient SQL query.

It turns out the GraphQL library provides a way of doing these sort of optimizations, where we want to "look ahead" at the rest of the GraphQL query and perform more efficient resolution calls. It may not be appropriate to do this in every case, but certain products and workloads may benefit tremendously.

Open up `server.js` and let's do a bit of work on the `node` field's `resolve` function. Aside from `source` and `args`, GraphQL passes two more variables to `resolve`: `context` and `info`. We'll use `context` later to help with authentication and authorization; `info` is a bag of objects, including an



*abstract syntax tree (AST)* of the entire GraphQL query. We'll do a simple traversal of the AST and determine if we should run a more efficient loader:

```
    node: {
      type: NodeInterface,
      args: {
        id: {
          type: new GraphQLNonNull(GraphQLID)
        }
      },
      resolve(source, args, context, info) {
        let includeFriends = false;

        const selectionFragments = info.fieldASTs[0].selectionSet.selections;
        const userSelections = selectionFragments.filter((selection) => {
          return selection.kind === 'InlineFragment' && selection.typeCondition.\
name.value === 'User';
        })

        userSelections.forEach((selection) => {
          selection.selectionSet.selections.forEach((innerSelection) => {
            if (innerSelection.name.value === 'friends') {
              includeFriends = true;
            }
          });
        });

        if (includeFriends) {
          return loaders.getUserNodeWithFriends(args.id);
        }
        else {
          return loaders.getNodeById(args.id);
        }
      }
    }
```

There's a lot happening as we traverse the AST, and we won't go into detail on most of the specifics. If you end up performing these look-ahead optimizations in your code, you can `console.log` each level of the tree and determine what information you can access.

Essentially we look for `User` fragments on the `node` field's selection set, and determine if the fragment accesses the `friends` field. If it does, then we run a new loader; else, we fall back to the original loader.

Now let's take a look at the new `loaders.getUserNodeWithFriends` function:

```
export const getUserNodeWithFriends = (nodeId) => {
  const { tableName, dbId } = tables.splitNodeId(nodeId);

  const query = tables.users
    .select(tables.usersFriends.user_id_b, tables.users.star())
    .from(
      tables.users.leftJoin(tables.usersFriends)
      .on(tables.usersFriends.user_id_a.equals(tables.users.id))
    )
    .where(tables.users.id.equals(dbId))
    .toQuery();


  return database.getSql(query).then((rows) => {
    if (!rows[0]) {
      return undefined;
    }

    const __friends = rows.map((row) => {
      return {
        user_id_b: row.user_id_b,
        __tableName: tables.users.getName()
      }
    });

    const source = {
      id: rows[0].id,
      name: rows[0].name,
      about: rows[0].about,
      __tableName: tableName,
      __friends: __friends
    };
    return source;
  });
};
```

This starts getting a bit complicated, and very specific to this product and the frameworks we chose - which is a common pattern when working on performance optimizations. Our SQL query now grabs all of the friends and the user's profile simultaneously, eliminating a database round-trip. We then load those friends into a `__friends` property (we've chosen to continue the __ prefix for "private" properties), and can access it inside of `types.js`:



```
    friends: {
      type: new GraphQLList(GraphQLID),
      resolve(source) {
        if (source.__friends) {
          return source.__friends.map((row) => {
            return tables.dbIdToNodeId(row.user_id_b, row.__tableName);
          });
        }

        return loaders.getFriendIdsForUser(source).then((rows) => {
          return rows.map((row) => {
            return tables.dbIdToNodeId(row.user_id_b, row.__tableName);
          });
        })
      }
    }
```

Other applications might perform look-ahead optimizations differently - instead of making globbing multiple SQL queries into one, they might warm a cache in the background. The important takeaway is that `resolve` accepts many arguments which let you short-circuit the usual recursive resolution flow.

## Lists Continued

Our `friends` field returns a complete list of IDs (performantly, might I add), but what we really want is a list to their full `User` types. For large lists, we'd probably want to use an idiomatic GraphQL connection (which we'll implement soon) but we'll allow the `friends` field to return all entries on each query.

We'll start by removing the logic we added for performance optimization. Since our application is about to change a bit, we can revisit performance once its capabilities have stabilized.

```
      resolve(source, args, context, info) {
        return loaders.getNodeById(args.id);
      }
```

Next we need to change the `type` returned by our `friends` field to a list of `User`. Since we already have a loader for loading arbitrary nodes by ID, there's not much code we need to write:

```
export const UserType = new GraphQLObjectType({
  name: 'User',
  interfaces: [ NodeInterface ],
  // Note that this is now a function
  fields: () => {
    return {
      id: {
        type: new GraphQLNonNull(GraphQLID),
        resolve: resolveId
      },
      name: { type: new GraphQLNonNull(GraphQLString) },
      about: { type: new GraphQLNonNull(GraphQLString) },
      friends: {
        type: new GraphQLList(UserType),
        resolve(source) {
          return loaders.getFriendIdsForUser(source).then((rows) => {
            const promises = rows.map((row) => {
              const friendNodeId = tables.dbIdToNodeId(row.user_id_b, row.__tabl\
eName);
              return loaders.getNodeById(friendNodeId);
            });
            return Promise.all(promises);
          })
        }
      }
    };
  }
});
```

We now set the `friends` type to `GraphQLList(UserType)`. Because of how JavaScript variable hoisting works, we have to change the `fields` property to a function instead of an object in order to pick up a "recursive" type definition (where a type returns a field of itself). We invoke `loaders.getNodeById` on all of the IDs we previously retrieved and voila! Restart your server and execute this kind of query in GraphiQL:



```
{
  node(id:"users:4") {
    ... on User {
      friends {
        id
        about
        name
      }
    }
  }
}
```

Which should return this data:

```
{
  "data": {
    "node": {
      "friends": [
        {
          "id": "users:1",
          "about": "Sports!",
          "name": "Harry"
        },
        {
          "id": "users:3",
          "about": "Love books",
          "name": "Hannah"
        },
        {
          "id": "users:2",
          "about": "I'm the best",
          "name": "David"
        }
      ]
    }
  }
}
```

You can even go a step further and query the `friends` of `friends`!

## Connections

Now we want to implement idiomatic connection fields. Instead of returning a simple list, we're going to return a more complicated (but powerful) structure. Although there is additional work, connections fields are most appropriate for lists that would otherwise be large or unbounded. It might be prohibitive or wasteful to return a huge list in one query, so GraphQL schemas prefer to break up these fields into smaller paginated chunks.

Instead of using literal page numbers, recall from the last chapter that idiomatic GraphQL uses opaque strings called *cursors*. Cursors are more resilient to real-time changes to your data, which might lead to duplicates in simple page-based systems. The `pageInfo` field of a connection gives metadata to help with making new requests, while the `edges` field will hold the actual data for each item.

Instead of the previous query which used lists for `friends`, we want something like this for `posts`:

```
{
  node(id:"users:1") {
    ... on User {
      posts(first: 1) {
        pageInfo {
          hasNextPage
          hasPreviousPage
          startCursor
          endCursor
        }
        edges {
          cursor
          node {
            id
            body
          }
        }
      }
    }
  }
}
```

Instead of just returning a list of `PostType`, the `posts` field will now return a `PostsConnection` type.

Define the `PageInfo`, `PostEdge`, and `PostsConnection` types in your `types.js`, in addition to importing more types from the `graphql` library:



```
import {
  GraphQLInterfaceType,
  GraphQLObjectType,
  GraphQLID,
  GraphQLString,
  GraphQLNonNull,
  GraphQLList,
  GraphQLBoolean,
  GraphQLInt,
} from 'graphql';


const PageInfoType = new GraphQLObjectType({
  name: 'PageInfo',
  fields: {
    hasNextPage: {
      type: new GraphQLNonNull(GraphQLBoolean)
    },
    hasPreviousPage: {
      type: new GraphQLNonNull(GraphQLBoolean)
    },
    startCursor: {
      type: GraphQLString,
    },
    endCursor: {
      type: GraphQLString,
    }
  }
});

const PostEdgeType = new GraphQLObjectType({
  name: 'PostEdge',
  fields: () => {
    return {
      cursor: {
        type: new GraphQLNonNull(GraphQLString)
      },
      node: {
        type: new GraphQLNonNull(PostType)
      }
    }
  }
});
```

```
65
66  const PostsConnectionType = new GraphQLObjectType({
67    name: 'PostsConnection',
68    fields: {
69      pageInfo: {
70        type: new GraphQLNonNull(PageInfoType)
71      },
72      edges: {
73        type: new GraphQLList(PostEdgeType)
74      }
75    }
```

These are mostly just type definitions with no inherent `resolve` functions for now. Different applications will have different ways and patterns for resolving connections, so don't consider some of the implementation details here as the gospel for your own work.

Now we need to hook up our `UserType` to the new types we created and actually create the `posts` field.

```
100         }
101       },
102       posts: {
103         type: PostsConnectionType,
104         args: {
105           after: {
106             type: GraphQLString
107           },
108           first: {
109             type: GraphQLInt
110           },
111         },
112         resolve(source, args) {
113           return loaders.getPostIdsForUser(source, args).then(({ rows, pageInfo \
114 }) => {
115             const promises = rows.map((row) => {
116               const postNodeId = tables.dbIdToNodeId(row.id, row.__tableName);
117               return loaders.getNodeById(postNodeId).then((node) => {
118                 const edge = {
119                   node,
120                   cursor: row.__cursor,
121                 };
122                 return edge;
123               });
```



```
124             });
125
126             return Promise.all(promises).then((edges) => {
127               return {
128                 edges,
129                 pageInfo
130               }
131             });
132
133           })
134         }
135       }
136     };
137   }
138 });
```

This should look familiar to how we implement our `friends` field, aside from the new arguments. Remember that this field does not return a list of `PostType` - it returns a `PostsConnectionType`, which is an object with `pageInfo` and `edges` keys.

We use a new `loader` method, `getPostIdsForUser`, and pass it the `args` to our resolver. We'll implement this loader very soon, and it will not only return the associated `rows` but also a `pageInfo` structure that corresponds to the `PageInfoType`. We then load the nodes for each of the identifiers and create the wrap them into a `PostEdgeType` with the row's cursor.

There are ways to make this more efficient at the JavaScript and database levels, but for now let's focus on making our code work by implementing `getPostIdsForUser`.

This loader will determine what data to fetch based on the pagination arguments and what cursors to assign the rows returned. The algorithm for slicing and pagination through your data based on the arguments is rather complex when supporting all possibilities, and you can read about it in detail in the Relay specification[130]. For brevity, we're only going to support the `after` and `first` arguments.

We start by defining the new function and parsing the arguments:

```
74  export const getPostIdsForUser = (userSource, args) => {
75    let { after, first } = args;
76    if (!first) {
77      first = 2;
78    }
```

In other words, if the user does not supply an argument for `first`, then we will return two posts. Then we start to construct our SQL query:

[130] https://facebook.github.io/relay/graphql/connections.htm#sec-Pagination-algorithm

```
  const table = tables.posts;
  let query = table
    .select(table.id, table.created_at)
    .where(table.user_id.equals(userSource.id))
    .order(table.created_at.asc)
    .limit(first + 1);
```

We grab `first + 1` rows as a cheap method to determine if there are any more rows beyond what the user wants. Our query is ordered by `created_at ASC`, which is important in order to get deterministic data upon successive queries.

Next we account for an `after` cursor that may be passed:

```
  if (after) {
    // parse cursor
    const [id, created_at] = after.split(':');
    query = query
      .where(table.created_at.gt(after))
      .where(table.id.gt(id));
```

Our cursors in this example are strings composed of row IDs and row dates. Generally cursors will be based upon some date in most systems, since keeping IDs as incrementing integers is less common when working with high scale data.

We can finally execute our database query:

```
  return database.getSql(query.toQuery()).then((allRows) => {
    const rows = allRows.slice(0, first);

    rows.forEach((row) => {
      row.__tableName = tables.posts.getName();
      row.__cursor = row.id + ':' + row.created_at;
```

Remember that we actually queried one more row than the user requested, which is why we have to `slice` the rows returned. We also construct the cursor for each row and set the `__tableName` property so that our future `JOIN` queries will work.

Now that we have our rows, we execute it and start to create our `pageInfo` object:



```
    const hasNextPage = allRows.length > first;
    const hasPreviousPage = false;

    const pageInfo = {
      hasNextPage: hasNextPage,
      hasPreviousPage: hasPreviousPage,
    };

    if (rows.length > 0) {
      pageInfo.startCursor = rows[0].__cursor;
      pageInfo.endCursor = rows[rows.length - 1].__cursor;
    }
```

Keeping a reference to `allRows` lets us calculate `hasNextPage`; because we don't support the `before` and `last` arguments, we always set `hasPreviousPage` to false. Setting `startCursor` and `endCursor` is as simple as grabbing the first and last elements of our `rows` array.

We return both the `rows` and `pageInfo` objects to finish the loader - finally! Restart your server and try the query we described at the start of the section:

```
{
  node(id:"users:1") {
    ... on User {
      posts(first: 1) {
        pageInfo {
          hasNextPage
          hasPreviousPage
          startCursor
          endCursor
        }
        edges {
          cursor
          node {
            id
            body
          }
        }
      }
    }
  }
}
```

In response you should get the first post by this user:

```
{
  "data": {
    "node": {
      "posts": {
        "pageInfo": {
          "hasNextPage": true,
          "hasPreviousPage": false,
          "startCursor": "1:2016-04-01",
          "endCursor": "1:2016-04-01"
        },
        "edges": [
          {
            "cursor": "1:2016-04-01",
            "node": {
              "id": "posts:1",
              "body": "The team played a great game today!"
            }
          }
        ]
      }
    }
  }
}
```

See that `endCursor`? Now try running a query with that cursor as the `after` value:

```
{
  node(id:"users:1") {
    ... on User {
      posts(first: 1, after:"1:2016-04-01") {
        pageInfo {
          hasNextPage
          hasPreviousPage
          startCursor
          endCursor
        }
        edges {
          cursor
          node {
            id
            body
          }
```



```
        }
      }
    }
  }
}
```

This returns the next (and final, judging from `hasNextPage`) post in the series:

```
{
  "data": {
    "node": {
      "posts": {
        "pageInfo": {
          "hasNextPage": false,
          "hasPreviousPage": false,
          "startCursor": "2:2016-04-02",
          "endCursor": "2:2016-04-02"
        },
        "edges": [
          {
            "cursor": "2:2016-04-02",
            "node": {
              "id": "posts:2",
              "body": "Honestly I didn't do so well at yesterday's game, but eve\
ryone else did."
            }
          }
        ]
      }
    }
  }
}
```

Congratulations, you've now implemented cursor-based pagination! You might be able to use simple lists for static data, but using cursors prevents all kinds of frontend bugs and complexity for data that updates relatively often. It also allows you to leverage Relay's understanding of pagination and build paginated or infinite-scrolling UIs very quickly.

## Authentication

Earlier we noted that in our social network the friendships have "levels," which posts should respect. For example, if a post has a level of `friend`, then only my friends with a level of `friend` or higher (instead of `acquaintance` or a lower level) should see it.

This topic is generally referred to as *authorization*. GraphQL has no inherit notion or opinions on authorization, which makes it quite flexible for implementing controls on who can see what data in your schemas. This also means you need to take care to ensure that youâ€™re not accidentally exposing data that should be hidden to the user.

We're going to add a small *authentication* layer to our server, which verifies that the GraphQL query is allowed to be processed, as well as the authorization logic to control who can see the different posts. The techniques we'll use are definitely not the only ways to implement these features with GraphQL, but should spark some ideas that might apply to your product.

For authentication, we're going to use HTTP basic authentication[131]. There are a myriad of protocols for authentication, such as OAuth, JSON web tokens, and cookies, and the choice is ultimately very unique to each product. HTTP basic auth is fairly simple to add to our current NodeJS server, which is the primary reason in this case.

First, install the `basic-auth-connect` package, which provides a very simple API to allow certain credentials access:

```
$ npm i basic-auth-connect@1.0.0 --save --save-exact
```

Then in our server code, import the module:

```
3  import express from 'express';
4  import basicAuth from 'basic-auth-connect';
5
6  const app = express();
```

Right before we add our GraphQL endpoint, add a new call to `app.use`. Remember that Express will trigger each `app.use` function in the order they are added - if we put the new `basicAuth` function *after* our `graphqlHTTP` function, the ordering would be incorrect.

```
67 app.use(basicAuth(function(user, pass) {
68   return user === 'harry' && pass === 'mypassword1';
69 }));
70
71 app.use('/graphql', graphqlHTTP({ schema: Schema, graphiql: true }));
```

For now, we'll allow any user with the right password. Restart your server and try running this cURL command to test a simple query:

[131] https://en.wikipedia.org/wiki/Basic_access_authentication



```
$ curl -XPOST -H 'content-type:application/graphql' http://localhost:3000/graphq\
l -d '{ node(id:"users:4") { id } }'
Unauthorized
```

Since we didn't specify a username or password, our query fails. Try this next command to correctly pass our credentials:

```
$ curl -XPOST -H 'content-type:application/graphql' --user 1:mypassword1 http://\
localhost:3000/graphql -d '{ node(id:"users:4") { id } }'
{"data":{"node":{"id":"users:4"}}}
```

Great, now our authentication is working. You can also try this in Chrome and Firefox, which allow a GUI for entering the username and password.

The important concept here is that authentication is generally decoupled from a GraphQL schema. It's definitely possible to pass the username and password into a GraphQL query to authenticate the user (over HTTPS of course), but idiomatic GraphQL tends to separate the concerns.

## Authorization

Now, onto tackling authorization. Remember in the last chapter we brought up the idea of a `viewer` field, which represents the logged-in users node in the data graph. We're going to add that field to our schema and allow our resolution code to be aware of the viewer's permissions.

By using `basic-auth-connect`, we can access to a `user` property on every Express request. The specifics on how you determine the user making each request will vary depending on your authentication library, but we simple need to take that `request.user` property and forward to our GraphQL resolver:

```
77 app.use('/graphql', graphqlHTTP((req) => {
78   const context = 'users:' + req.user;
79   return { schema: Schema, graphiql: true, context: context, pretty: true };
80 }));
```

Instead of always returning the same `schema` and `graphiql` settings for all requests, we now return a different configuration object for each GraphQL query. This new configuration has the `context` property set to the username, like the `user1` from the example earlier.

The next question is how do we access that `context` inside our GraphQL fields? Recall from earlier in the chapter that each `resolve` function gets passed some arguments. We've become very familiar with the `args` argument, but it turns out the `context` is also passed.

Here's how we add the `viewer` field with that knowledge:

```
const RootQuery = new GraphQLObjectType({
  name: 'RootQuery',
  description: 'The root query',
  fields: {
    viewer: {
      type: NodeInterface,
      resolve(source, args, context) {
        return loaders.getNodeById(context);
      }
    },
```

If you restart your server, you should be able to cURL the endpoint like so:

```
$ curl -XPOST -H 'content-type:application/graphql' --user 1:mypassword1 http://\
localhost:3000/graphql -d '{ viewer { id } }'
{
  "data": {
    "viewer": {
      "id": "users:1"
    }
  }
}
```

You can explore using the `... on User` inline fragment to query more properties. We are able to provide this sort of consistent API with minimal code changes because we modeled our application data as a graph - neat!

Not only can top-level `viewer` field access the `context`, but *all* `resolve` functions have access, regardless of their depth in the hierarchy. This makes it very simple to add authorization checks to our `posts` field.

We start by forwarding the `context` argument in our `resolve` function:

```
        resolve(source, args, context) {
          return loaders.getPostIdsForUser(source, args, context).then(({ rows, \
pageInfo }) => {
```

GraphQL schemas should not handle authorization logic directly, which is likely to duplicate logic from your main codebase. Instead, that responsibility should fall into the underlying data loading libraries or services, as we show here.

Inside our `getPostIdsForUser`, we need to load each post's `level` from the database before we can use it. All we have to do is add it to our `select` arguments:



```
  let query = table
    .select(table.id, table.created_at, table.level)
    .where(table.user_id.equals(userSource.id))
    .order(table.created_at.asc)
    .limit(first + 10);
```

In addition to running the database query to get all of the posts, we're also going to run another query to get all of the user access levels for our `context`. We'll use that list of levels to filter down the results our database query.

```
  return Promise.all([
    database.getSql(query.toQuery()),
    getFriendshipLevels(context)
  ]).then(([ allRows, friendshipLevels ]) => {
    allRows = allRows.filter((row) => {
      return canAccessLevel(friendshipLevels[userSource.id], row.level);
    });
    const rows = allRows.slice(0, first);
```

We're referencing two new functions that have yet to be implemented, `getFriendshipLevels` and `canAccessLevel`. Before we get to that, note that this does potentially introduce a bug into our system. We were calculating `hasNextPage` based on the length of `allRows`, but now `allRows` can be truncated depending on privacy settings. This highlights some of the complexity of systems that are highly aware of authorization; a naive mitigation of this is to just read more rows eagerly from the database, which we changed above (`first + 10`).

The `getFriendshipLevels` definition is similar to our other queries:

```
const getFriendshipLevels = (nodeId) => {
  const { dbId } = tables.splitNodeId(nodeId);

  const table = tables.usersFriends;
  let query = table
    .select(table.star())
    .where(table.user_id_a.equals(dbId));

  return database.getSql(query.toQuery()).then((rows) => {
    const levelMap = {};
    rows.forEach((row) => {
      levelMap[row.user_id_b] = row.level;
    });
    return levelMap;
  });
};
```

At the end we transform the array of rows into an object for a slightly more efficient API (you can also implement this transformation using a single `reduce` function if you'd like).

The last piece is `canAccessLevel`. Because our privacy settings are totally linear, we can represent the settings as an array and use the indices as a simple comparison:

```
const canAccessLevel = (viewerLevel, contentLevel) => {
  const levels = ['public', 'acquaintance', 'friend', 'top'];
  const viewerLevelIndex = levels.indexOf(viewerLevel);
  const contentLevelIndex = levels.indexOf(contentLevel);

  return viewerLevelIndex >= contentLevelIndex;
};
```

That all wasn't too bad, was it? We can test this out with some queries. Login as user `1` (so username `1` and password `mypassword1`) and run this query:

```
{
  node(id:"users:2") {
    ... on User {
      posts {
        edges {
          node {
            id
            ... on Post {
              body
            }
          }
        }
      }
    }
  }
}
```

In return, you'll see no posts. This is because our `context` (user 1) is not friends with the node we're accessing (user 2), and their posts have a level of `friend`.

Now open an incognito window, login as user `5`, run that same query. You'll see a post!



```
{
  "data": {
    "node": {
      "posts": {
        "edges": [
          {
            "node": {
              "id": "posts:3",
              "body": "Hard at work studying for finals..."
            }
          }
        ]
      }
    }
  }
}
```

This is because user `5` is actually a friend of user `2`.

This is a simple example, but highlights a few points about GraphQL.

- GraphQL server libraries typically allow you to forward on some kind of query-level `context`
- Your GraphQL schema code should not concern itself with authorization logic, instead deferring to your underlying data code

We've been reading data from our server for a bit, but now we should try out changing some of it with mutations.

## Rich Mutations

There's only so much your application can do if it can only read data from the server - more often than not, we have to upload some new data. Recall from the last chapter that in GraphQL, we call these updates *mutations*.

We're going to add a mutation to our schema which creates a new post. The field will have a string arguments for the post body and a friendship privacy level, and it will allow us to query more information about the resulting post object.

Earlier in this chapter we added a simple key-value mutation in our `server.js` file. Let's update that definition to use the arguments and types we expect for creating a new post:

```
import { GraphQLSchema, GraphQLObjectType, GraphQLString,
  GraphQLNonNull, GraphQLID, GraphQLEnumType } from 'graphql';


const LevelEnum = new GraphQLEnumType({
  name: 'PrivacyLevel',
  values: {
    PUBLIC: {
      value: 'public'
    },
    ACQUAINTANCE: {
      value: 'acquaintance'
    },
    FRIEND: {
      value: 'friend'
    },
    TOP: {
      value: 'top'
    }
  }
});

const RootMutation = new GraphQLObjectType({
  name: 'RootMutation',
  description: 'The root mutation',
  fields: {
    createPost: {
      type: PostType,
      args: {
        body: {
          type: new GraphQLNonNull(GraphQLString)
        },
        level: {
          type: new GraphQLNonNull(LevelEnum),
        }
      },
      resolve(source, args, context) {
        return loaders.createPost(args.body, args.level, context).then((nodeId) \
=> {
          return loaders.getNodeById(nodeId);
        });
      }
    }
```



```
  }
});
```

First we instantiate a new kind of object, a `GraphQLEnumType`. We only briefly mentioned the `Enum` GraphQL type in previous chapter, but it works similarly to how enums work in many programming languages. Because our `level` argument should only be one of a fixed amount of options, we enforce that contract at the schema level with an enum. By convention, enums in GraphQL are `ALL_CAPS`.

After creating our enum, we use it in the `args` property of our new `createPost` mutation. Note that in addition to the arguments, `createPost` has a `type` of `PostType`, which means we eventually need to return a post object after performing our mutating code. That work is actually deferred to a new `createPost` loader, which looks like this:

```
export const createPost = (body, level, context) => {
  const { dbId } = tables.splitNodeId(context);
  const created_at = new Date().toISOString().split('T')[0];
  const posts = [{ body, level, created_at, user_id: dbId }];

  let query = tables.posts.insert(posts).toQuery();
  return database.getSql(query).then(() => {
    return database.getSql({ text: 'SELECT last_insert_rowid() AS id FROM posts'\
 });
  }).then((ids) => {
    return tables.dbIdToNodeId(ids[0].id, tables.posts.getName());
  });
};
```

This is mostly specific to our SQLite database, but you can imagine how this would work in other frameworks or data stores. We construct our database row, insert it, and then retrieve the newly inserted ID.

If you open up GraphiQL, you should be able to give this mutation a try:

```
mutation {
  createPost(body:"First post!", level:PUBLIC) {
    id
    body
  }
}
```

In the wild, you may run into more complex scenarios for updating data such as uploading files. The specifics will depend on what server language and library you use, but it's supported with

Relay and GraphQL-JS. The Relay documentation[132] discusses how files are handled, and you can find examples elsewhere[133] of how to leverage those within your GraphQL schema.

## Relay and GraphQL

The "Facebook-lite" schema we've developed may be small, but it should give you an idea of how to structure common operations in a GraphQL server. It also happens to be compatible with Relay[134], Facebook's frontend React library for working with a GraphQL server.

In addition to publishing Relay itself, Facebook publishes a library to help you more easily construct a Relay-compatible GraphQL server with Node. This GraphQL-Relay-JS[135] package reduces much of the boilerplate we've experienced, especially for some of Relay's more powerful features.

You should read over the docs for all the details, but we're briefly going to convert some of our code to use this library. First we need to install it via npm:

```
$ npm install graphql-relay@0.4.1 --save --save-exact
```

GraphQL-Relay is particularly helpful with connection fields. Although we only had one proper connection field in our schema (`posts`), you can imagine that repeating the types and code for each connection in a larger app would become tiresome. Luckily, all we need is a quick import:

```
15 import {
16   connectionDefinitions
17 } from 'graphql-relay';
```

Then delete all of our existing connection types, so that we skip straight to the `UserType`:

```
34 const resolveId = (source) => {
35   return tables.dbIdToNodeId(source.id, source.__tableName);
36 };
37
38 export const UserType = new GraphQLObjectType({
39   name: 'User',
```

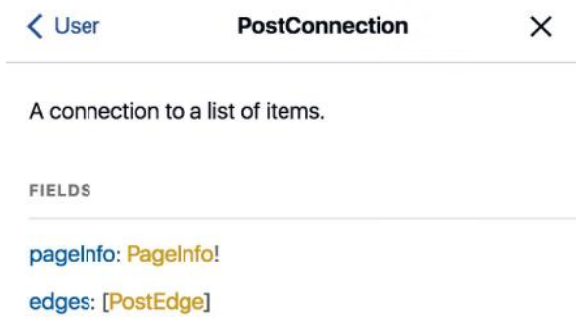
And at the very bottom, add this one-liner to define `PostsConnectionType`:

[132] https://facebook.github.io/relay/docs/api-reference-relay-mutation.html#getfiles
[133] http://stackoverflow.com/a/35585482
[134] https://facebook.github.io/relay/
[135] https://github.com/graphql/graphql-relay-js



```
116 const { connectionType: PostsConnectionType } = connectionDefinitions({ nodeType\
117 : PostType });
```

Internally, this generates all the types we crafted by hand - `PageInfo`, `PostEdge`, and `PostConnection`. You can confirm this if you load up the GraphiQL documentation:




GraphQL Relay

In addition to connections, the GraphQL-Relay library also has functions for simplifying how node types are structured[136] and working with Relay-compatible mutations[137]. We'll explore this more in the upcoming Relay chapter, but Relay imposes some rules on how mutations work, similar to the way that certain types and arguments required for connections.

None of the GraphQL server changes required for Relay are specific to GraphQL-JS or JavaScript in general. Any language GraphQL server can be made compatible with Relay, and hopefully this chapter has made you more familiar with the basic building blocks of any GraphQL schema.

## Performance: N+1 Queries

Now that our schema has settled, we can reconsider performance. Before we dive in, remember that the performance needs of different products can be incredibly disparate. Engineers should carefully consider the costs and benefits of writing code that is more performant at the risk of additional complexity.

Let's consider a query like this:

[136] https://github.com/graphql/graphql-relay-js#object-identification
[137] https://github.com/graphql/graphql-relay-js#mutations

```
{
  node(id:"users:4") {
    ... on User {
      friends {
        edges {
          node {
            id
            about
            name
          }
        }
      }
    }
  }
}
```

Under the hood, our current GraphQL resolution code will trigger one database query to get the node for `"users:4"`, one query to get the list of friend IDs, and N database queries for each of the `friends` edges. This is commonly referred to as the N+1 Query Problem[138] and can easily occur using any web framework or ORM. You can imagine that for a slow database or a large number of edges, this will cause degraded performance. We can visualize this with the following raw SQL:

```
SELECT "users".* FROM "users" WHERE ("users"."id" = 4) LIMIT 1
SELECT "users_friends"."user_id_b" FROM "users_friends" WHERE ("users_friends"."\
user_id_a" = 4)
SELECT "users".* FROM "users" WHERE ("users"."id" = 1) LIMIT 1
SELECT "users".* FROM "users" WHERE ("users"."id" = 3) LIMIT 1
SELECT "users".* FROM "users" WHERE ("users"."id" = 2) LIMIT 1
```

In a perfect world, we would only need two database queries: one to retrieve the initial node, and then one to retrieve **all** of the friends in one query (or within whatever paging limits we need) - in other words, we batch all of the queries for friends. Something more like this would not:

```
SELECT "users".* FROM "users" WHERE ("users"."id" = 4) LIMIT 1
SELECT "users_friends"."user_id_b" FROM "users_friends" WHERE ("users_friends"."\
user_id_a" = 4)
SELECT "users".* FROM "users" WHERE ("users"."id" in (1, 3, 2)) LIMIT 3
```

Consider how loading a user works: it's a call to `loaders.getNodeById`. Currently that function immediately triggers a database query, but what if we could "wait" for some small amount of

[138] https://secure.phabricator.com/book/phabcontrib/article/n_plus_one/



time, collect node IDs that need to be loaded, and then trigger a database query like the one above? GraphQL and JavaScript enable intuitive techniques for batching alike queries, which we'll implement.

Facebook maintains a library called DataLoader[139] to help, which is a generic JavaScript library independent of React or GraphQL. You use the library to create *loaders*, which are objects that automatically batch fetching of similar data. For example, you would instantiate a `UserLoader` to load users from the database:

```
const UserLoader = new DataLoader((userIds) => {
  const query = table
    .select(table.star())
    .where(table.id.in(userIds))
    .toQuery();

  return database.getSql(query.toQuery());
});

// elsewhere, loading a single user
function resolveUser(userId) {
  return UserLoader.load(userId);
}
```

Notice how `UserLoader.load` takes a single `userId` as an argument, but the argument to it's internal function is an array of `userIds`. This means if we call `UserLoader.load` from multiple places in rapid succession, we have the option of crafting a more efficient database query.

In our app, most of our code touches `loaders.getNodeById` which makes it a strong candidate for automatic batching. None of the code that invokes `getNodeById` has to change; instead, we're going to internally batch node fetches using DataLoader.

First, install DataLoader from npm:

```
$ npm install dataloader@1.2.0 --save --save-exact
```

Now onto our changes to `loaders.js`. We're going to make one data loader per table, which is a reasonable way to start optimizing. The code starts like this:

[139] https://github.com/facebook/dataloader

```
import * as database from './database';
import * as tables from './tables';

import DataLoader from 'dataloader';

const createNodeLoader = (table) => {
  return new DataLoader((ids) => {
    const query = table
      .select(table.star())
      .where(table.id.in(ids))
      .toQuery();

    return database.getSql(query).then((rows) => {
      rows.forEach((row) => {
        row.__tableName = table.getName();
      });
      return rows;
    });
  });
};
```

Our `createNodeLoader` is a factory function which returns a new instance of `DataLoader`. We craft a query of the form `SELECT * FROM $TABLE WHERE ID IN($IDS)`, which lets us select multiple nodes with a single query.

Now we need to invoke our factory function, which we'll store in a constant:

```
const nodeLoaders = {
  users: createNodeLoader(tables.users),
  posts: createNodeLoader(tables.posts),
  usersFriends: createNodeLoader(tables.usersFriends),
};
```

Finally, we change our definition of `getNodeById` to use the appropriate loader:

```
export const getNodeById = (nodeId) => {
  const { tableName, dbId } = tables.splitNodeId(nodeId);
  return nodeLoaders[tableName].load(dbId);
};
```

If you open up GraphiQL and try this query, you'll notice the SQL logs in the server console are appropriately batching our database fetches:



```
{
  user3: node(id:"users:3") {
    id
  }
  user4: node(id:"users:4") {
    id
  }
}
```

```
$ node index.js
{ starting: true }
{ running: true }
SELECT "users".* FROM "users" WHERE ("users"."id" IN ($1, $2))
[ '3', '4' ]
```

Note that our higher-level GraphQL schema code is totally unaware of this optimization and didn't have to change at all. In general, you should prefer keeping optimizations at the loader and data service level so all consumers can enjoy the benefits.

DataLoader is simple but powerful tool. Although we showed off its batching abilities, it can also act as a cache - if you'd like to learn more, check out its documentation[140] and consider watching this talk by its maintainer[141].

## Summary

We covered a lot of ground in this chapter. We designed a schema, created a GraphQL server from scratch, connected it to a relational database, and explored some performance optimizations. Regardless of your production language and stack, these concepts will apply to all GraphQL server implementations. Additionally, this should give you some more understanding and context whenever you connect to a GraphQL server from the frontend.

We used the Facebook-maintained GraphQL-JS library in this chapter, but the GraphQL ecosystem is exploding. Here some more popular options and technologies you might want to explore:

- GraphQL[142] for Ruby
- Graphene[143] for Python
- Sangria[144] for Scala

[140] https://github.com/facebook/dataloader#getting-started
[141] https://www.youtube.com/watch?v=OQTnXNCDywA
[142] https://github.com/rmosolgo/graphql-ruby
[143] https://github.com/graphql-python/graphene
[144] https://github.com/sangria-graphql/sangria

- Apollo Server[145] for Node
- Graffiti-Mongoose[146] for Node and MongoDB
- Services like Reindex[147] and Graphcool[148] for hosting GraphQL servers

Now that we've learned about consuming GraphQL and authoring a GraphQL server, it's time to put everything we've covered thus far and learn about Relay.

[145] http://docs.apollostack.com/apollo-server/tools.html
[146] https://github.com/RisingStack/graffiti-mongoose
[147] https://www.reindex.io/
[148] https://graph.cool/

# Relay

## Introduction

Picking the right data architecture in a client-server web app can be difficult. There are a myriad of decisions to be made when you try to implement a data architecture. For example:

- How will we fetch data from the server?
- What do we do if that fetch fails?
- How do we query dependencies and relationships between data
- How do we keep data consistent across the app?
- How can we keep developers from breaking each others' components accidentally?
- How can we move faster and write our app-specific functionality rather than spending our time pushing data back and forth from our server?

Thankfully, Relay has a theory on all of these issues. Even better: it's an implementation of that theory and it's a joy to use, once you get it set up. But it's natural to ask: What *is* Relay?

Relay is a library which connects your React components to your API server.

We talked about GraphQL earlier in the book and it may not be immediately clear what the relationship is between Relay, GraphQL, and React. You can think of Relay as the glue between GraphQL and React.

Relay is great because:

- components declare the data they need to operate
- it manages *how* and *when* we fetch data
- it intelligently aggregates queries, so we don't fetch more than we need
- it gives us clear patterns for navigating relationships between our objects and mutating them

One of the most helpful things about Relay that we'll see in this chapter is that the GraphQL queries are **co-located with the component**. This gives us the benefit of being able to see a clear specification of the values needed to render a component directly alongside the component itself.

Each component specifies what it needs and Relay figures out if there's any duplication, before making the query. Relay will aggregate the queries, process the responses, and then hand back to each component just the data it asked for.

GraphQL can be used independent of Relay. You can, in theory, create a GraphQL server to have more-or-less any schema you want. However, Relay has a specification that your GraphQL server must follow in order to be Relay-compatible. We'll talk about those constraints.

## What We're Going to Cover

In this chapter we're going to walk through a practical tutorial on **how to setup and use Relay in a client app**.

Relay depends on a GraphQL server, which we've provided in the sample code, but we're not going to talk about the implementation details of that server.

In this chapter we're going to:

- explain the various concepts of Relay
- describe how to setup Relay in our app (with routing)
- show how to fetch data from Relay into our components
- show how to update data on our server using mutations
- highlight tips and tricks about Relay along the way

By the end of this chapter you'll understand what Relay is, how to use it, and have a firm foundation for integrating Relay into your own apps.

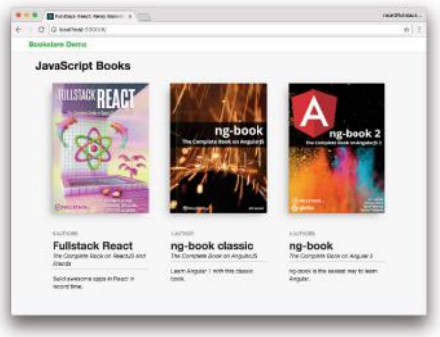
## What We're Building

### The Client

In this chapter, we're going to build a simple bookstore. We'll have three pages:

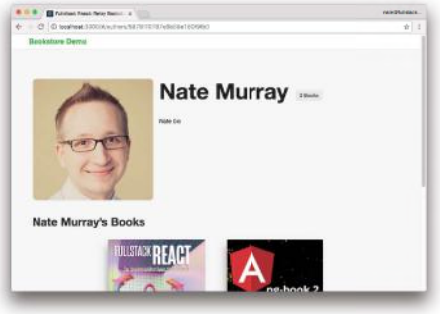
- A book listing page, which shows all of the books we have for sale.





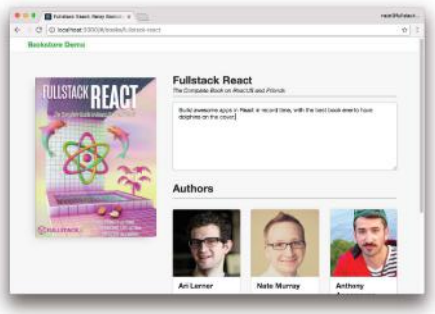

Books Store Page

- An author bio page, which shows an author's information and the books they've authored.




Author Page

- A book listing page, which shows a single book and the authors for that book. We'll also **edit** a book on this page.




**Book Listing Page**

The data model here is straightforward: a book has (and belongs to) many authors.

This data is provided to our app via an API server.

## The Server

We've provided a Relay-compatible GraphQL demo server in the sample code with this app. In this chapter **we aren't going to discuss the implementation details of the server**. If you're interested in building GraphQL servers, checkout our chapter on GraphQL Servers.

This server was created with the library graffiti-mongoose[149]. If you're using MongoDB and mongoose[150], then `graffiti-mongoose` is a fantastic choice for adapting your models to GraphQL. `graffiti-mongoose` takes your `mongoose` models and automatically generates a Relay-compatible GraphQL server.

That said, there are libraries that can help you generate GraphQL from your models for every popular language. Checkout the list awesome-graphql[151] to see what libraries are available.

## Try The App

You can find the project for this chapter inside the `relay` folder of the code download:

[149] https://github.com/RisingStack/graffiti-mongoose
[150] http://mongoosejs.com/
[151] https://github.com/chentsulin/awesome-graphql



```
$ cd relay/bookstore
```

We've included the completed server and completed client. Before we can run them, we need to run `npm install` on both the client and the server:

```
$ npm install
$ cd client
$ npm install
$ cd ..
```

Next, you can run them individually in two separate tabs with:

```
# tab one
$ npm run server
# tab two
$ npm run client
```

Or we've provided a convenience command that will run them both with:

```
$ npm start
```

When they're running, you should be able to access the GraphQL server at `http://localhost:3001` and the client app at `http://localhost:3000`.

In this chapter, we're going to be spending time in our browser looking at **both the client and the server**. We've installed the GraphQL GUI tool "GraphiQL" on this demo server.

Relay is based on GraphQL, and so to get a better understanding of how it works, we're going to be running a few queries "by hand" in this GraphiQL interface.

## How to Use This Chapter

While we will *introduce* all of the terminology of Relay, this chapter is intended to be a guide to working with Relay and not the API documentation.

As you're reading this chapter, feel free to cross-reference the official docs[152] to dig in to the details.

### Prerequisites

This is an advanced chapter. We're going to focus on using Relay and not re-cover the basics of writing React apps. It's assumed that you're somewhat comfortable with writing components, using props, JSX, and loading third party libraries. Relay also requires GraphQL, and so we assume you have some familiarity with GraphQL as well.

If you don't feel comfortable with these things, we cover all of them earlier in the book.

[152] https://facebook.github.io/relay/

### Relay 1

Currently this chapter covers Relay 1.x. There is a new version of Relay in the works, but don't let that stop you from learning Relay 1. There is no public release date and Facebook employee Jaseph Savona stated on GitHub[153] that Relay 2 has "definite API differences, but the core concepts are the same and the API changes serve to make Relay simpler and more predictable"

Some good news here is that the underlying GraphQL specification doesn't change[154]. So while, yes, Relay will have a v2 in the future. We can still use Relay 1 in our production apps today.

If you're interested more in the future of Relay, checkout Relay: State of the State[155].

## Guide to the Code Structure

In this chapter, we've provided the **completed** version of the app in `relay/bookstore`.

In order to break up the concepts in to more digestible bites, we've broken up some of the components into steps. We've included these intermediate files in `relay/bookstore/client/steps`.

Our Relay app contains a server, a client, and several build tools – these all add up to quite a lot of files and directories. Here's a brief overview of some of the directories and files you'll find in our project. (Don't worry if some of this is unfamiliar, we'll explain everything you need to know in this chapter):

```
-- bookstore
    |-- README.md
    |-- client
    |   |-- config                      // client configuration
    |   |   |-- babelRelayPlugin.js     // our custom babel plugin
    |   |   |-- webpack.config.dev.js   // webpack configuration
    |   |   `-- webpack.config.prod.js
    |   |-- package.json
    |   |-- public                      // images and index.html
    |   |   |-- images/
    |   |   `-- index.html
    |   |-- scripts                     // helper scripts
    |   |   |-- build.js
    |   |   |-- start.js
    |   |   `-- test.js
```

[153] https://github.com/facebook/relay/issues/1369
[154] https://github.com/facebook/relay/issues/1369#issuecomment-266647767
[155] https://facebook.github.io/react/blog/2016/08/05/relay-state-of-the-state.html



```
|   `-- src
|       |-- components                 // our components are here
|       |   |-- App.js
|       |   |-- AuthorPage.js
|       |   |-- BookItem.js
|       |   |-- BookPage.js
|       |   |-- BooksPage.js
|       |   |-- FancyBook.js
|       |   `-- TopBar.js
|       |-- data                        // graphql metadata
|       |   |-- schema.graphql
|       |   `-- schema.json
|       |-- index.js
|       |-- mutations                   // mutations make changes
|       |   `-- UpdateBookMutation.js
|       |-- routes.js                   // our routes
|       |-- steps/                      // intermediate files
|       `-- styles                      // css styles here
|-- models.js                           // server models
|-- package.json
|-- schema.js                           // graphql schema on the server
|-- server.js                           // our server definition
|-- start-client.js
|-- start-server.js
`-- tools
    `-- update-schema.js                // a helper for generating the schema
```

Feel free to poke around at the sample code we've provided, but don't worry about understanding every file just yet. We'll cover all of the important parts.

## Relay is a Data Architecture

Relay is a JavaScript library that runs on the client-side. You connect Relay to your React components and then Relay will fetch data from your server. Of course, your server also needs to conform to Relay's protocol, and we'll talk about that, too.

Relay is designed to be the data-backbone for your React app. This means that, ideally, we'd use Relay for all of our data loading and for maintaining the central, authoritative state for our application.

Because Relay has its own store it's able to cache data and resolve queries efficiently. If you have two components that are concerned with the same data, Relay will combine the two queries into one and then distribute the appropriate data to each component. This has the duel benefit of minimizing the

number of server calls we need but still allowing the individual components the ability to specify locally what data they need.

Because Relay holds a central store of your data, this means it's not really compatible with other data architectures that keep a central store, like Redux. You can't have two central stores of state and so this makes the current versions of Redux and Relay essentially incompatible.

**Apollo**

If you're already using Redux but you'd like to try Relay, there's still hope: the Apollo project[156] is a Relay-inspired library that is based on Redux. If you're careful, you can retrofit Apollo into your existing Redux app.

We're not going to talk about Apollo in this chapter, but it's a fantastic library and it's absolutely worth looking into for your app.

## Relay GraphQL Conventions

One thing we need to clarify is the relationship between Relay and GraphQL.

Our GraphQL server defines our GraphQL schema and how to resolve queries against that schema. GraphQL itself lets you define a wide variety of schemas. For the most part, it doesn't dictate what the structure of your schema should be.

**Relay defines a set of conventions on top of GraphQL**. In order to use Relay, your GraphQL server must adhere to a specific set of guidelines.

At a high level, these conventions are:

1. A way to fetch **any object by ID** (regardless of type)
2. A way to **traverse relationships** between objects (called "connections") with pagination
3. A structure around changing data with **mutations**

These three requirements allow us to build sophisticated, efficient apps. In this chapter, we'll make these generic guidelines concrete.

Let's explore implementations of these three Relay conventions directly on our GraphQL server. Then later in the chapter we'll implement them in our client app.

**Relay Specification Official Docs**

You can checkout Facebook's Official Documentation on the Relay/GraphQL specification here[157]

[156] http://www.apollodata.com/

[157] https://facebook.github.io/relay/docs/graphql-relay-specification.html



## Exploring Relay Conventions in GraphQL

Make sure you have the GraphQL server running, as described above, and open it in your browser at the address `localhost:3001`.

Remember that GraphiQL reads our schema and provides a documentation explorer to navigate the types. Click on the "Docs" link in GraphiQL to show the "Root Types" of our schema.

GraphiQL Interface with Docs

## Fetching Objects By ID

In our server we have two models: `Author` and `Book`. The first thing we're going to do is look-up a specific object by ID.

Imagine that we want to create a page that shows the information about a particular author. We might have a URL such as `/authors/abc123`, where `abc123` is the ID of the author. We'd want Relay to ask our server "what is the information for the author `abc123`?" In that case, we'd use a GraphQL query like we're going to define below.

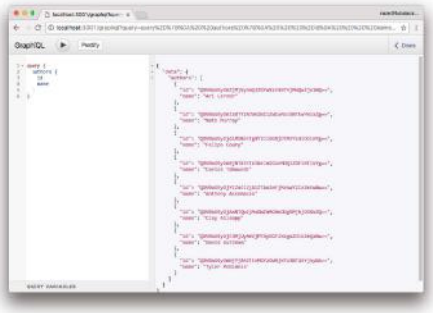
However, we have a bit of a chicken and egg problem at the moment: we don't know the IDs of any individual record.

So let's load the entire list of authors' `names` and `ids` and then take note of one of the IDs.

Enter the following query into GraphiQL:

```
query {
  authors {
    id
    name
  }
}
```

And click the play button:



**GraphiQL Authors with IDs**

Now that we have an author ID, we can query `author` to get the author with a *specific* id:

Query:

```
query {
  author(id: "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==") {
    id
    name
  }
}
```

Response:



```
{
  "data": {
    "author": {
      "id": "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==",
      "name": "Anthony Accomazzo"
    }
  }
}
```

While this is handy, from the `author` query we can only receive an object of type `Author`, so it doesn't fulfill the requirement of the Relay specification. The Relay specification says that we need to have a way to query **any object** (`Node`) via the `node` query.

We'll talk more about our GraphQL schema later in the chapter, but it's worth pointing out that both `Author` and `Book` types implement the `Node`-type interface.

We've implemented the ability to lookup a `Node` via `node` on our server, so let's try it out. First, let's query just the `id` from `node`:

Query:

```
query {
  node(id: "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==") {
    id
  }
}
```

Response:

```
{
  "data": {
    "node": {
      "id": "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw=="
    }
  }
}
```

This works! But it isn't very useful because we don't have any other data about the author. Let's try fetching the name:

Query:

```
query {
  node(id: "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==") {
    id
    name
  }
}
```

This fails with the error:

```
Cannot query field "name" on type "Node". Did you mean to use an inline fragment\
 on "Author" or "Book"?
```

What happened here? Because `Node` is a generic type, we can't query the `name` field. Instead we need to provide a *fragment* which says, if we're querying an `Author`, then return `Author` specific fields. So we can adjust our query like so:

Query:

```
query {
  node(id: "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==") {
    id
    ... on Author {
      name
    }
  }
}
```

Response

```
{
  "data": {
    "node": {
      "id": "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==",
      "name": "Anthony Accomazzo"
    }
  }
}
```

It works! We're able to fetch an author using their `ID`. The key idea here is that we can query **any object** in our system by using the `node` query by ID (which is a Relay requirement). For instance, if we had a `Book` ID, we could also look up a book using the `node` query as well.



A useful exercise would be to try looking up a `Book` by ID using `node`. Hint: You'll need to add a fragment `on Book` using the syntax: `... on Book` and then specifying the `Book` fields you wish to retrieve.

**Globally Unique IDs**

The implications here are that we actually have a globally unique ID for each object. If, for instance, you were using a traditional SQL database (like Postgres) with auto-incrementing IDs per-table, you may have an `Author` with ID `2` and a `Book` with ID `2`. How do we resolve this?

This issue is resolved by the GraphQL server. The idea is that you'll have to come up with a convention that, say, embeds the table name and numeric ID into a GUID. Then your GraphQL server would encode and decode these IDs.

This actually is happening with the server we've provided. You may have noticed that our schema models have both an `id` field and an `_id` field. What's the difference?

The `_id` field is the MongoDB ID. The `id` field is the *Relay GUID*, a composite of the table name and the `_id` field.

Relay uses the `node` interface particularly for *re-fetching objects*. When writing our apps, we can have dozens of ways of loading various objects. The idea behind the `node` interface is to give a consistent, easy way for Relay to ask the server "what's the most current value for this object, given this ID?"

Now that we can query individual `Nodes`, we need to talk about how we traverse relationships *between* them. In our example app we're going to show a list of books on the homepage, and we want to be able to load the authors who wrote that book.

In Relay we will indicate a relationship between an `Author` and `Book` by using a *connection*.

## Walking Connections

An `Author` may have contributed to several `Books`.

If you're familiar with traditional relational databases, maybe you've seen this relationship modeled with:

- An `authors` table which has an `id`
- A `books` table which has an `id`
- and an `authorships` table which holds both an `author_id` and a `book_id`

The idea in this scenario would be that for every `Author`/`Book` pair, you'd create this new "join model" called an `Authorship` which represents an `Author` who contributed to a particular `Book`. This idea is sometimes referred to as "has-and-belongs-to-many".

Analogously, Relay also defines a "join model" that should be used to denote a relationship between two models. To be precise, Relay specifies **two**:

1. The "connection" model which specifies a relationship between two models and keeps pagination data and
2. The "edge" model which wraps a cursor as well as the node- (model-) specific data.

This might seem a bit of overkill the first time you come to it, but recognize that this is a powerful and flexible model that provides consistent pagination across your models.

**What's a *cursor*?**

When we're traversing a large set of items, we need to keep track of where we are in that traversal. For instance, imagine that we're doing pagination through a list of products. The simplest "cursor" could be the current **page number** (e.g. page number `3`).

Of course, in real applications you may have items being added and removed to the list all the time and so you might find that by the time you load page `4` you miss some records (because some records have been added or deleted).

So what can we do in this case? This is where a *cursor* comes in. A *cursor* is a value take indicates where we "are" in traversing a list. Implementations vary, but the idea is that you can send the cursor to the server and the server give you the next page of results.

For example Twitter's API uses cursoring and you can checkout their docs here[158]

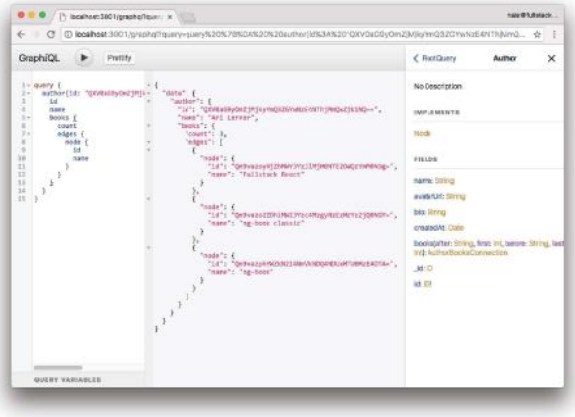
Let's try a few queries where we traverse these connections.

Here's a query that will get an author along with all of their book names:

```
query {
  author(id: "QXV0aG9yOmZjMjkyYmQ3ZGYwNzE4NThjMmQwZjk1NQ==") {
    id
    name
    books {
      count
      edges {
        node {
          id
          name
        }
      }
    }
  }
}
```

[158] https://dev.twitter.com/overview/api/cursoring





**GraphiQL Author with Books**

Click through the documentation for `Author` in the GraphiQL interface. Notice that on the `Author` type `books` doesn't return an array of `Book`, but rather it:

1. accepts arguments, such as `first` or `last` and
2. Returns an `AuthorBooksConnection`


**GraphiQL Author Type**

Relay was designed to handle relationships that have a huge number of items. In the case where we have too many `Books` to load at once, we could use these arguments to limit the number of results that returned. We use the `first` and `last` arguments to denote the number of items we want to retrieve.

The `AuthorBooksConnection` has three fields:

- `pageInfo`
- `edges` and
- `count`

`count` gives us the total number of `edges` we have (the number of `Books` this person has authored, in this case).

`pageInfo` gives us information such as if there is a previous or next page and what the start and end cursor for this page are. We could use these cursors later on to ask for the next (or previous) page.

`edges` contains the list of `AuthorBooksEdge`. Look at the `AuthorBooksEdge` type in the GraphiQL interface.




**GraphiQL `AuthorBooksEdge` Type**

The `AuthorBooksEdge` has two fields:

- `node` and
- `cursor`

The `cursor` is a string we can use for pagination, from this record. Here we can see that the `node` is of type `Book`. Within that `node` is the hard data that we're looking for.

While we are using these connections for many-to-many relationships, you'd navigate one-to-many relationships the same way. By using this standard for traversing relationships between models it's easy to know how to get access to related models, regardless of the type of relationship. Everything is nodes and edges in a graph.

Additionally, by making pagination part of the standard, we're making accommodations for real-world data sets where we're not going to load every piece of data in a relationship (or table) at the same time.

Having standardized pagination as a part of Relay make it that much easier to standardize pagination in our React apps.

**Which part is Relay and which part is GraphQL?**

Relay vs. GraphQL with connections and edges

To be clear, here we're talking about Relay specification that we implement on our GraphQL server.

The pattern of connections and edges, with the fields defined above, is part of the Relay specification. Anytime we need a relationship between two objects (that goes beyond a simple array) we'll use the edge/connection paradigm.

Relay specifies the convention (e.g. that we specify `first` or `last` when retrieving a connection, and a connection returns `edges`) and our GraphQL server implements how we actually fetch those records (from the database).

## Changing Data with Mutations

Mutations are how we change data in Relay/GraphQL. To change data with mutations we need to:

1. Locate the mutation we want to call
2. Specify the input arguments and
3. Specify what data we want to be returned after the mutation is finished.

Say, for instance, that we want to change an author's bio. We could perform the following:

Query:

```
mutation {
  updateAuthor(input: {
    id: "QXV0aG9yOmZjMjkyYmQ3ZGYwNzE4NThjMmQwZjk1NQ==",
    bio: "all around great guy"
  }) {
    changedAuthor {
      id
      name
      bio
    }
  }
}
```

Now we can see in the response that the `bio` has been changed to `"all around great guy"`.

Response:



```
{
  "data": {
    "updateAuthor": {
      "changedAuthor": {
        "id": "QXV0aG9yOmZjMjkyYmQ3ZGYwNzE4NThjMmQwZjk1NQ==",
        "name": "Ari Lerner",
        "bio": "all around great guy"
      }
    }
  }
}
```

In this case, the *mutation* is `updateAuthor`. As you may recall from the GraphQL chapter, we change data in GraphQL by creating a *mutation query*. Our server defines a mutation query for common data operations.

For instance, if you're familiar with the REST paradigm of Create-Read-Update-Delete (CRUD), we might define similar mutations for our models here: `createAuthor`, `updateAuthor`, `deleteAuthor`.

While mutations are used generally in GraphQL, Relay, unsurprisingly, specifies some constraints around how it expects *mutations* to be defined.

Mutations on the client-side can be a bit daunting. We'll talk more about them later in the chapter.

## Relay GraphQL Queries Summary

Above, we've just covered the three types of queries we'll use when writing Relay apps:

1. Fetching individual records from our server
2. Traversing relationships between records, using connections and edges
3. Mutating data with mutation queries.

Now that we've explored our data and reviewed the conventions on our server, let's take a look at how Relay works inside our app.

# Adding Relay to Our App

## Quick Look at the Goal

Now let's switching gears into writing our React app.

There are quite a few steps to installing Relay into our app. Before we walk through them, let's take a look at the goal: a basic Relay container component that loads data from our server and renders it.

Once we have one component that can load data from Relay, building our app gets much easier from there.

Below we'll walk through how to build out a fully working Relay app, but it can also be helpful to take a look at a **single-file, "hello world" example**. Below is the code for a minimal, but working Relay app. Many of the specifics will be unfamilar, and the rest of the chapter is dedicated to explaining in detail how to use each idea. For now, just skim this code to get an idea of the different parts involved in working with Relay.

**relay/bookstore/client/src/steps/index.minimal.js**

```
/* eslint-disable react/prefer-stateless-function */
import React from 'react';
import ReactDOM from 'react-dom';
import Relay from 'react-relay';

import '../semantic/dist/semantic.css';
import './styles/index.css';

// Customize this based on your server's URL
const graphQLUrl = 'http://localhost:3001/graphql';

// Configure Relay with a "NetworkLayer"
Relay.injectNetworkLayer(
  new Relay.DefaultNetworkLayer(graphQLUrl)
);

// Create the top-level query that we'll execute
class AppQueries extends Relay.Route {
  static routeName = 'AppQueries';
  static queries = {
    viewer: () => Relay.QL`
      query {
        viewer
      }
    `,
  };
}

// A basic component that renders the list of authors
class App extends React.Component {
  render() {
    return (
      <div>
```



```
        <h1>Authors list</h1>
        <ul>
          {this.props.viewer.authors.edges.map(edge =>
            <li key={edge.node.id}>{edge.node.name}</li>
          )}
        </ul>
      </div>
    );
  }
}

// A Relay Container that specifies the fragment to be used in our query above
const AppContainer = Relay.createContainer(App, {
  fragments: {
    viewer: () => Relay.QL`
      fragment on Viewer {
        authors(first: 100) {
          edges {
            node {
              id
              name
            }
          }
        }
      }
    `,
  },
});

ReactDOM.render(
  <Relay.Renderer
    environment={Relay.Store}
    Container={AppContainer}
    queryConfig={new AppQueries()}
  />,
  document.getElementById('root')
);
```

In the example above, our `AppContainer` fetches a collection of authors and renders them in a list. Notice that this code is *declarative*. That is, there's no part of the code where we're saying "make a post request to the server and interpret it as JSON" etc. Instead, our component declares the data it needs, and Relay fetches it from the server and provides it to our component.

## A Preview of the Author Page

Let's look at another example, taken from our Bookstore app. Here's a version of the `AuthorPage` component:

**relay/bookstore/client/src/steps/AuthorPage.minimal.js**

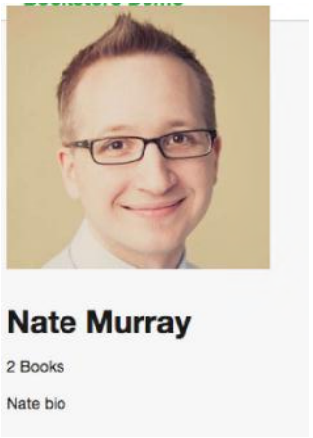
```
8  class AuthorPage extends React.Component {
9    render() {
10     const { author } = this.props;
11
12     return (
13       <div>
14         <img src={author.avatarUrl} />
15         <h1>{author.name}</h1>
16         <p>
17           {/* e.g. '2 Books' or '1 Book' */}
18           {author.books.count}
19           {author.books.count > 1 ? ' Books' : ' Book'}
20         </p>
21         <p>{author.bio}</p>
22       </div>
23     );
24   }
25 }
26
27 export default Relay.createContainer(AuthorPage, {
28   fragments: {
29     author: () => Relay.QL`
30     fragment on Author {
31       name
32       avatarUrl
33       bio
34       books {
35         count
36       }
37     }`,
38   },
39 });
```

There are two parts: 1. the `Relay.createContainer` statement (with a `Relay.QL` query) and 2. the `render()` function.



At a high level, what's happening here is that our `Relay.QL` query specifies the data we want to load for this author. Then the `author` is passed in to the component via `props` and we `render` the data.

We haven't yet talked about all of the setup that goes in to getting our app to this point. We will. For now, just notice that once we have everything setup, it becomes extremely easy to load data into our components (and **it's easy to modify our queries if we change our mind**).




**Minimal Author**

Notice that when we use `Relay` on our component, we have two things: 1. the `createContainer` and 2. a `fragment`.

## Containers, Queries, and Fragments

Components that need to load data from Relay are called *containers*. Containers specify *fragments* that are essentially "partial queries". Fragments **specify the data this component needs to render properly**.

We create a Relay container using `Relay.createContainer`. We pass in our component and the required fragments as arguments.

Your containers specify the `fragments` that they need to render properly but you have to **execute the queries to render the fragments**. You can think of this sort of like components needing to be rendered into the DOM. **Fragments aren't rendered until they're pulled in by a query.**

Each container specifies a fragment (or several) and at some point later we execute a query which will use this fragment to fetch data.

That is, before this data will become available to the component, you have to **execute a query that contains this fragment**. We'll talk about executing queries in a few minutes.

But first, let's talk about the `Relay.QL` query.

## Validating Our Relay Queries at Compile Time

One of the great things about GraphQL is that we have a type-schema for our API. We can use this to our advantage when writing client side code.

When we write query fragments in our components, it looks like this:

code/relay/bookstore/client/src/steps/AuthorPage.minimal.js

```
  fragments: {
    author: () => Relay.QL`
    fragment on Author {
      name
      avatarUrl
      bio
      books {
        count
      }
    }`,
  },
```

But notice that we're putting the query in a long JavaScript string. It's easy to write, but in a naive implementation of handling these queries, typos could be the source of many bugs because there's no way to validate that the contents are well formed. In the case where something was mis-typed, we wouldn't even know there was a typo until one of these queries was run.

However, the good news is that there is a custom Babel plugin which will verify these queries against our schema **at compile time**.

For this we'll use the `babel-relay-plugin`[159]. This plugin:

1. reads these `Relay.QL` backtick strings
2. parses the GraphQL query
3. validates them against our schema
4. and converts our `Relay.QL` queries to a expanded function calls[160]

But in order to verify our schema, **we need to have the schema available to our client-side build tools**.

Remember that our schema is **defined by our server**. It's our GraphQL *server* that defines the data models (Books and Authors, in this case) and the corresponding GraphQL schema.

So how do we make our server's schema available to our client build-tools? **We export the schema from the server as a JSON file and copy it over.**

[159] https://www.npmjs.com/package/babel-relay-plugin
[160] This is similar to how JSX works



## Building the `schema.json`

To provide the schema to our client build tools we'll write a script **in the server** that will export the schema to a JSON file.

We'll then configure `babel-relay-plugin` to use this JSON file when we compile our client code. It's a little bit of work up-front, but the benefit is that we'll get compile-time validation of or Relay queries in our client app. This can save our team tons of time trying to track down bugs because of invalid queries.

Here's the script we're using to generate our `schema.json`:

relay/bookstore/tools/update-schema.js

```
import fs from 'fs';
import path from 'path';
import { graphql } from 'graphql';
import { introspectionQuery, printSchema } from 'graphql/utilities';
import schema from '../schema';

// Save JSON of full schema introspection for the Babel Relay Plugin to use
const generateJSONSchema = async () => {
  var result = await (graphql(schema, introspectionQuery));
  if (result.errors) {
    console.error(
      'ERROR introspecting schema: ',
      JSON.stringify(result.errors, null, 2)
    );
  } else {
    fs.writeFileSync(
      path.join(__dirname, '../client/src/data/schema.json'),
      JSON.stringify(result, null, 2)
    );
  }

  // Save user readable type system shorthand of schema
  fs.writeFileSync(
    path.join(__dirname, '../client/src/data/schema.graphql'),
    printSchema(schema)
  );
};

generateJSONSchema().then(() => {
  console.log("Saved to client/src/data/schema.{json,graphql}");
})
```

Take a look at the dependencies we're loading here. Besides `fs` and `path`, we have:

1. `graphql`, `introspectionQuery`, `printSchema` from `graphql` and
2. `schema` from `../schema`.

One thing to notice is that we're not loading anything from the `relay` library. This is all GraphQL. The `relay` library isn't involved. Our GraphQL schema *conforms* to the Relay standard, but we don't depend on any Relay-specific functionality. This process of exporting your schema can be done with **any** GraphQL server.

We aren't going to look deeply into the implementation of our `schema`. In this case, we're using the helper library `graffiti-mongoose`, but that's really an implementation detail. This `schema` can be any `GraphQLSchema` object.

So to use this script for your app, simply change the paths to the `schema` as well as the output paths.

What's happening here is that we're telling the `graphql` library to take our `schema` and then `introspectionQuery` and print out our schema into two files:

The `introspectionQuery` is a query which asks GraphQL information about what queries it supports. You can read more about GraphQL Introspection here[161]

1. a machine-readable `schema.json` file (for our client) and
2. a human-readable `schema.graphql` file

Here's a sample from the `schema.graphql` file:

**relay/bookstore/client/src/data/schema.graphql**

```
type Author implements Node {
  name: String
  avatarUrl: String
  bio: String
  createdAt: Date
  books(after: String, first: Int, before: String, last: Int): AuthorBooksConnec\
tion
  _id: ID

```

[161] http://graphql.org/learn/introspection/



```
  # The ID of an object
  id: ID!
}

# A connection to a list of items.
type AuthorBooksConnection {
  # Information to aid in pagination.
  pageInfo: PageInfo!

  # A list of edges.
  edges: [AuthorBooksEdge]
  count: Float
}
```

If we're in the root directory for this project (`relay`) then we can generate this schema by running:

```
npm run generateSchema
```

If we ever change our schema (like adding a field to a model or adding a new model) then we need to run this script to regenerate our schema. If we don't regenerate the schema and try to use the new data in our client app then `babel-relay-plugin` will throw compiler errors because it will using the old schema. So make sure if you change your models, you regenerate your schema.

**Watch out for schema caching**

Some webpack configurations (such as the default that is generated when you eject from `create-react-app`) cache compiled scripts. If you're using such a feature (as we are in this app) then your **schema is also cached**.

This means that when you update your schema, you have to clear your `react-scripts` cache. On my machine, this folder is kept in `node_modules/.cache/react-scripts`. So whenever we regenerate the schema we run the following command as well to clear the cache:

```
rm -rf client/node_modules/.cache/react-scripts/
```

Failure to clear this cache when regenerating your schema may result in your client loading the old, cached schema which can be confusing.

## Installing `babel-relay-plugin`

To use `babel-relay-plugin` we have to:

1. `npm install babel-relay-plugin`
2. Tell `babel-relay-plugin` about our `schema.json`
3. Configure `babel` to use our plugin

To do #2, see the file `client/config/babelRelayPlugin.js` like so:

**relay/bookstore/client/config/babelRelayPlugin.js**

```
var getBabelRelayPlugin = require('babel-relay-plugin');
var schema = require('../src/data/schema.json');

module.exports = getBabelRelayPlugin(schema.data, {
  debug: true,
  suppressWarnings: false,
  enforceSchema: true
});
```

What we're doing here is just configuring the `babel-relay-plugin` by adding in our own schema and setting a few options. Now we need to add this script as a plugin to our `babel` config.

To do this, we've added the following to the *client's* `package.json`:

```
"babel": {
  "presets": [
    "react-app"
  ],
  "plugins": [
    "./config/babelRelayPlugin"
  ]
},
```

Above, we've added `"./config/babelRelayPlugin"` to the `plugins` section of our `babel` configuration in the `package.json`.

When you're setting up your own app to use Relay, it's fine if you have a different set of `presets` and `plugins` in your app, just add this custom `babelRelayPlugin` to the list.

Also, if you configure babel via a `.babelrc` or some other way, the core idea here is to add our custom `babelRelayPlugin` script to the list of `plugins`.



## Setting Up Routing

Now that we have our build tools in place, we can start integrating Relay with React. Because we're building a multi-page app, we're going to need a router. For this app we're going to use `react-router` with `react-router-relay`.

If you'd like to see an example of a Relay app that uses Relay directly (without using `react-router` then checkout the `relay-starter-kit`[162].

`react-router-relay` uses `react-router` v2.8 (**not router v4**, which we covered in the Routing Chapter.

Unfortunately, as you can see here[163], there are no plans to directly support Relay in React-Router v4.

That said, it is possible to integrate Relay with any routing framework including React Router v4. Doing so is beyond the scope of this chapter.

In this chapter, we'll provide all of the route configuration needed to run the app, but we're not going to be discussing the React Router API . If you need to look it up, you can find the docs for Router here[164].

`react-router-relay` provides a convenient way to execute Relay queries whenever the route changes. `react-router-relay` also will read parameters from the URL and pass them as parameters to our Relay queries. We'll take a close look at this feature in a it.

To install `react-router-relay` into our app, we'll do the following:

1. Configure Relay
2. Configure our `Router`
3. Use the `react-router-relay` middleware to connect Relay to our `Router`.

Let's look at the code we use to do this:

[162] https://github.com/relayjs/relay-starter-kit
[163] https://github.com/relay-tools/react-router-relay/issues/193
[164] https://github.com/ReactTraining/react-router

**relay/bookstore/client/src/index.js**

```
import React from 'react';
import ReactDOM from 'react-dom';
import createHashHistory from 'history/lib/createHashHistory';
import Relay from 'react-relay';
import applyRouterMiddleware from 'react-router/lib/applyRouterMiddleware';
import Router from 'react-router/lib/Router';
import useRouterHistory from 'react-router/lib/useRouterHistory';
import useRelay from 'react-router-relay';

import routes from './routes';

import '../semantic/dist/semantic.css';
import './styles/index.css';

// Customize this based on your server's URL
const graphQLUrl = 'http://localhost:3001/graphql';

// Configure Relay with a "NetworkLayer"
Relay.injectNetworkLayer(
  new Relay.DefaultNetworkLayer(graphQLUrl)
);

const history = useRouterHistory(createHashHistory)({ queryKey: false });

ReactDOM.render(
  <Router
    history={history}
    routes={routes}
    render={applyRouterMiddleware(useRelay)}
    environment={Relay.Store}
  />,
  document.getElementById('root')
);
```

The first section `imports` our dependencies.

Next we configure the URL to our server in the variable `graphQLUrl`. If your server is at a different URL, configure it here. For instance, we'll often use an environment-specific variable here.

Next we configure Relay with a "Network Layer". In this case, we're using `DefaultNetworkLayer` which will make HTTP requests, but you could use this to, say, mock out a Relay server for testing or use a different protocol entirely.



We're going to use hash-based routing for this app, so we configure `history` to use `createHashHistory`.

We tie our `Router` root component to Relay by:

1. Using `applyRouterMiddleware(useRelay)` – which comes from `react-router-relay` and
2. Setting our `Relay.Store` onto the `Router` `environment`.

Setting up our Router this way makes Relay *available* to our app, but to actually perform Relay queries we have one more step: we need to configure Relay queries on our routes.

## Adding Relay to Our Routes

Let's take a look at our `routes.js`:

**relay/bookstore/client/src/steps/routes.author.js**

```
import Relay from 'react-relay';
import React from 'react';
import IndexRoute from 'react-router/lib/IndexRoute';
import Route from 'react-router/lib/Route';

import App from './components/App';
import AuthorPage from './components/AuthorPage';

const AuthorQueries = {
  author: () => Relay.QL`
  query {
    author(id: $authorId)
  }`,
};

export default (
  <Route
    path='/'
    component={App}
  >
    <Route
      path='/authors/:authorId'
      component={AuthorPage}
      queries={AuthorQueries}
    />
  </Route>
);
```

For these initial routes we have a **parent** route that uses the App component and a **child** route that uses the AuthorPage component. Eventually we will have a child component for each page in our app, but for now let's look at the following in order:

1. Our parent App Component
2. The AuthorQueries and how the relate to the AuthorPage then
3. Dig in to the AuthorPage component.

## App Component

Our top-level App component establishes the wrapper for the rest of the app:

**relay/bookstore/client/src/components/App.js**

```
2  import React, { Component } from 'react';
3  import { withRouter } from 'react-router';
4
5  import TopBar from './TopBar';
6  import '../styles/App.css';
7
8  class App extends React.Component {
9    render() {
10     return (
11       <div className='ui grid'>
12         <TopBar />
13         <div className='ui grid container'>
14           { React.cloneElement(this.props.children) }
15         </div>
16       </div>
17     );
18   }
19 }
20
21 export default withRouter(App);
```

Here we render the TopBar and the markup that will wrap any child components (`this.props.children`).

Before we `export App`, we wrap it using `withRouter` from `react-router`. `withRouter` is a helper function[165] that provides `props.router` on our component.

We *could* make the App component a Relay container if we needed to, but in this case we don't need any Relay data in the App component.

[165] https://github.com/ReactTraining/react-router/blob/master/docs/API.md#withroutercomponent-options



## AuthorQueries Component

Now that we understand the App component, hop back into `routes.js` and look at the AuthorQueries:

**relay/bookstore/client/src/steps/routes.author.js**

```
9  const AuthorQueries = {
10   author: () => Relay.QL`
11   query {
12     author(id: $authorId)
13   }`,
14 };
```

Remember that in Relay in order to fetch the data that we need for our components we have to execute *queries*. When using `react-router-relay` we specify that when a particular route is visited, **the queries defined on that route will be executed**.

Notice that the AuthorQueries has one query: `author`. This query also has a variable `$authorId`. Where does the `$authorId` variable come from? It comes from the route path parameter:

**relay/bookstore/client/src/steps/routes.author.js**

```
21     <Route
22       path='/authors/:authorId'
23       component={AuthorPage}
24       queries={AuthorQueries}
25     />
```

Here in this route we're saying that we'll match the route `/authors/` and whatever follows will be interpreted as the `authorId`. This `authorId` is passed as a variable to the Relay query.

Notice something else odd about the `author` query: it doesn't specify any "leaf nodes" of data to fetch. The query stops at `author(id: $authorId)`. This is because the query is leaving the decision of what particular data to fetch to the component.

It is in our **component** (well, our Relay Container, to be precise), that we will specify the fields that are needed to render that component.

With that in mind, let's turn our attention to the AuthorPage component and look at the Relay query there.

## AuthorPage Component

Here's the code that specifies what fields we need to render the minimal AuthorPage:

relay/bookstore/client/src/steps/AuthorPage.minimal.js

```
27 export default Relay.createContainer(AuthorPage, {
28   fragments: {
29     author: () => Relay.QL`
30     fragment on Author {
31       name
32       avatarUrl
33       bio
34       books {
35         count
36       }
37     }`,
38   },
39 });
```

On the "inside" of the query it's easy to see that we're asking for the `name`, `avatarUrl`, and `bio` of the author. We're even able to dip into the `books` relationship and get the `count` of the number of books this person has authored.

However, it may not be clear how this ties into the query above. There are two constraints we need to follow.

The first is that the key names of these `fragments` must match the keys names of the queries. In this case, because this component is being rendered with a query key name of `author` (in `AuthorQueries`) the fragment key name must also be `author` (in `AuthorPage` `fragments`).


Relay Fragment Naming

The second constraint is that **this fragment will be a `fragment on` the type of the inner field of `query`**. That is, we wouldn't put `fragment on Book` here, because the containing query "ends" on an `Author`. We can see this by looking at the GraphQL schema in GraphiQL.




Relay Fragment on Type

When a route matching `/authors/:authorId` is visited, the `AuthorPage` `author` fragment will be pulled into the `AuthorQueries` and results will be fetched from the server and passed down into our component.

## Try It Out

We've only setup the `AuthorPage` so far, but it's enough for us to try out.

To do this, ensure you have both the GraphQL server started, as well as the client, as described above.

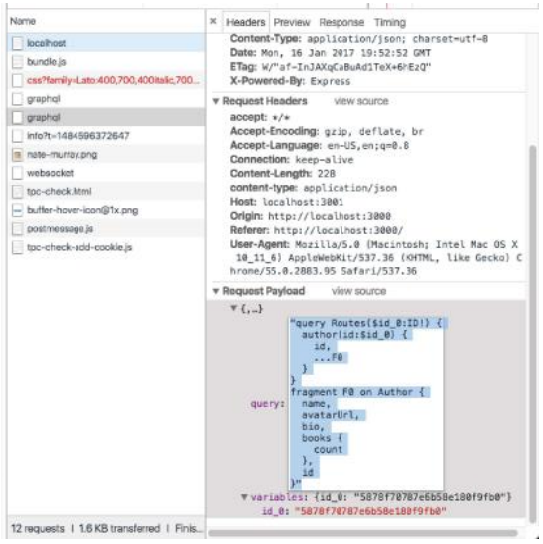
Then visit the GraphiQL server at `http://localhost:3001/graphql` and try the following query:

```
query {
  authors {
    id
    name
  }
}
```

Copy an author ID and then visit the client app with that ID. Something like:

```
http://localhost:3000/#/authors/QXV0aG9yOmIzOTY2NDVmZmZiZWIxMzc5NTEwYWIxZg==
```

In our browser, if we look at our network pane we can see the GraphQL query that was called:




**Relay GraphQL Call in Network Pane**

In this case, the query sent over the wire looks something like:

```
query Routes($id_0:ID!) {
  author(id:$id_0) {
    id,
    ...F0
  }
}
fragment F0 on Author {
  name,
  avatarUrl,
  bio,
  books {
    count
  },
  id
}
```



The first part of the query (`query Routes...`) comes from `AuthorQueries` and the fragment `F0` comes from our `author` fragment on `AuthorPage`. Again, note that it wouldn't make any sense for this fragment `F0` to be on any type *other than* `Author` because the child of `query > author` is of type `Author`. Trying to put a `Book`- or any other-typed fragment here would be invalid.

## `AuthorPage` with Styles

The minimal `AuthorPage` we've rendered so far doesn't look so great. Let's quickly add a bit of markup so that the page looks better.

> Like many of the examples in this book, we're using Semantic UI[166] for the CSS framework. When you see CSS `class` names like `sixteen wide column` or `ui grid centered` these are coming from Semantic UI.

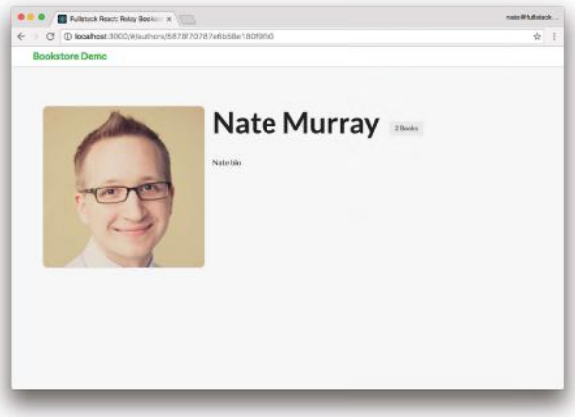
Keeping the same Relay query, we're going to change the `AuthorPage` markup to the following:

**relay/bookstore/client/src/steps/AuthorPage.styled.js**

```
8  class AuthorPage extends React.Component {
9    render() {
10     const { author } = this.props;
11
12     return (
13       <div className='authorPage bookPage sixteen wide column'>
14         <div className='spacer row' />
15
16         <div className='ui divided items'>
17           <div className='item'>
18             <div className='ui'>
19               <img src={author.avatarUrl}
20                 alt={author.name}
21                 className='ui medium rounded bordered image'
22               />
23             </div>
24             <div className='content'>
25               <div className='header authorName'>
26                 <h1>{ author.name }</h1>
27                 <div className='extra'>
28                   <div className='ui label'>
29                     { author.books.count }
30                     { author.books.count > 1 ? ' Books' : ' Book' }
```

[166] http://semantic-ui.com/

```
                  </div>
                </div>
              </div>
              <div className='description'>
                <p> { author.bio } </p>
              </div>

            </div>
          </div>
        </div>

      </div>
    );
  }
}
```



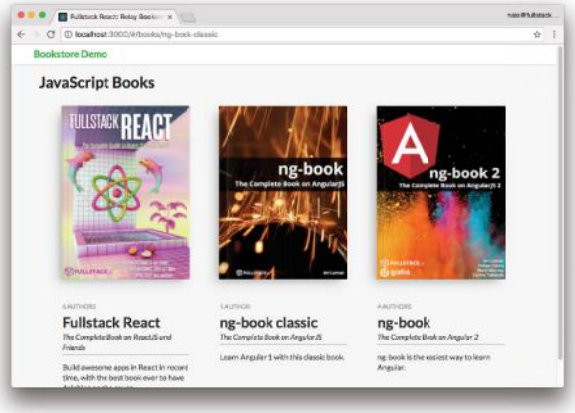

**Author Page with Styling**

We'll add the list of this author's books to this page later, but for now let's build the "index" page of the site, which will show the list of all of the books available.



## BooksPage

Here's what the list of books page will look like when we're finished:




**Books Page**

The first thing we need to do is create the route for the `BooksPage` and the queries for that route.

## `BooksPage` Route

The `BooksPage` is going to be the default page for our app so we will use the `IndexRoute` helper to define this route:

**relay/bookstore/client/src/routes.js**

```
    <IndexRoute
      component={BooksPage}
      queries={ViewerQueries}
    />
```

Let's take a look at the `ViewerQueries`:

relay/bookstore/client/src/routes.js

```
const ViewerQueries = {
  viewer: () => Relay.QL`query { viewer }`,
};
```

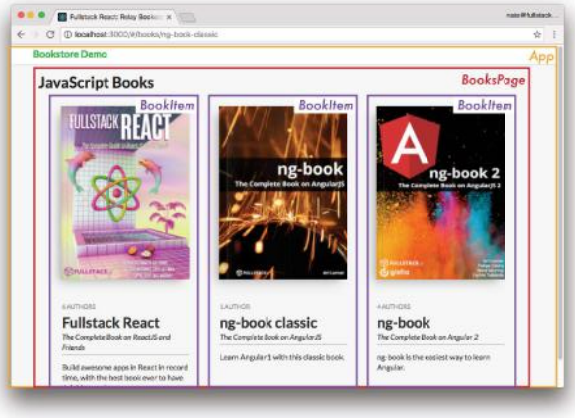
On this page, we're going to be looking up the list of books via the `viewer` node.

The `viewer` node is not strictly part of Relay but it's a pattern that you see in many GraphQL apps. The idea is that **the "viewer" is the current user of the app**. So, often in a real application you'd see that the `viewer` field accepts a user-identity field as an argument, such as an authentication token.

Imagine creating an app with a social feed. We could use the `viewer` field to get the feed for a particular user as opposed to the "firehose" feed for *all* users.

In this case, we're going to use the `Viewer` to load a list of `Books`.

## `BooksPage` Component




**Books Page Components**

Here we have **two** Relay containers:

1. the `BooksPage` which holds all the books
2. a `BookItem` which renders a particular book



Because our `BooksPage` route specified queries on `viewer` we're going to specify a fragment on the key `viewer` (on the `Viewer`-type). Let's take a look at the query:

relay/bookstore/client/src/components/BooksPage.js

```
export default Relay.createContainer(BooksPage, {
  initialVariables: {
    count: 100,
  },
  fragments: {
    viewer: () => Relay.QL`
    fragment on Viewer {
      books(first: $count) {
        count
        edges {
          node {
            slug
            ${BookItem.getFragment('book')}
          }
        }
      }
    }
    `,
  },
});
```

There's a couple of new things here:

1. `initialVariables`
2. Composing a fragment, using `getFragment()`

### Fragment Variables

We can set variables for our queries by using the `initialVariables` field. Here you can see that we're telling Relay we want to set `$count` to `100`. In the query you can see that we're asking for the first 100 books.

By using this `viewer` we can't say "I want **all** the books". The reason for this is that this server is designed for large apps and you generally don't want to load every single item in your database. Typically we would use pagination.

In this case though, we're just going to set the `count` to `100` because that's more than enough for this simple app. But say that we wanted to change the variable `count`. How would we do it?

You use `this.props.relay.setVariables` like this:

```
this.props.relay.setVariables({count: 2});
```

If we called the above function you would see that we load only 2 books instead of the whole set.

More generally, we can use Relay variables when we want our component to be able to change parameters about the query that's being executed.

If you forget to set the proper variables (e.g. no `first` field) when trying to get a list of items through a connection, you might get an error like the following: {line-numbers=off} Uncaught Error: Relay transform error `You supplied the `edges` field on a connection named `authors`, but you did not supply an argument necessary to do so. Use either the `find`, `first`, or `last` argument.` in file `code/relay/bookstore/client/src/components/BookPage.js`. Try updating your GraphQL schema if an argument/field/type was recently added.

## Fragment Composition

This is important: if you use a child Relay component you need to embed the child's fragment in the parent's query by using `getFragment()`.

Remember that fragments are parts of queries and they aren't realized until they're part of an executed query. If you want a child component like `BookItem` to be able to render each book, **you have to include the `BookItem` fragment in the `BooksPage` query**.

This takes a little getting used to, and we haven't even looked at the `render` function of `BooksPage` yet, so let's do that and come back to this idea of fragment composition when we have a bit more context.

### `BooksPage render()`

Here's the rendering for `BooksPage`:

**relay/bookstore/client/src/components/BooksPage.js**

```
8  class BooksPage extends React.Component {
9    render() {
10     const books = this.props.viewer.books.edges.map(this.renderBook);
11
12     return (
13       <div className='sixteen wide column'>
14         <h1>JavaScript Books</h1>
15         <div className='ui grid centered'>
16           { books }
```



```
17         </div>
18       </div>
19     );
20   }
21
22   renderBook(bookEdge) {
23     return (
24       <Link
25         to={`/books/${bookEdge.node.slug}`}
26         key={bookEdge.node.slug}
27         className='five wide column book'
28       >
29         <BookItem
30           book={bookEdge.node}
31         />
32       </Link>
33     );
34   }
35 }
```

Our `render` function has a basic wrapper. We're taking `this.props.viewer.books.edges` and calling `renderBook` on each **edge**.

Notice that we're not calling `renderBook` on the book itself (`node`), but on the `edge`.

Within `renderBook`, we render a:

1. `Link` to `/books/${bookEdge.node.slug}` with a
2. `BookItem` component, populating it with a book, found in `bookEdge.node`

Let's dig deeper into `BookItem` and then we'll pop back up to `BooksPage` to take a closer look at how Relay manages parent/child fragment values.

### `BookItem`

Here's our Relay Query for `BookItem`:

relay/bookstore/client/src/components/BookItem.js

```
export default Relay.createContainer(BookItem, {
  fragments: {
    book: () => Relay.QL`
    fragment on Book {
      name
      slug
      tagline
      coverUrl
      pages
      description
      authors {
        count
      }
    }
    `,
  },
});
```

We're defining a fragment with the key of `book` on type `Book`. This query is asking for basic book information like the `name`, the `slug`, the number of `pages`, and we also dip into `authors` to get the `count`.

Here's the component definition:

relay/bookstore/client/src/components/BookItem.js

```
class BookItem extends React.Component {
  render() {
    return (
      <div className='bookItem'>
        <FancyBook book={this.props.book} />

        <div className='bookMeta'>
          <div className='authors'>
            { this.props.book.authors.count }
            { this.props.book.authors.count > 1 ? ' Authors' : ' Author' }
          </div>
          <h2>{ this.props.book.name }</h2>
          <div className='tagline'>{ this.props.book.tagline }</div>
          <div className='description'>{ this.props.book.description }</div>
        </div>
```



```
      </div>
    );
  }
}
```

Inside the `div bookMeta` we show the basic information such as the number of `authors`, the book `name`, `tagline` and `description`.

We also pass `this.props.book` to a pure component `FancyBook`, which simply provides the markup for the fancy 3D CSS effect on the book.

### `BookItem Fragment`

What's interesting about the `BookItem` fragment `book` is that **it isn't tied to a particular query**. Remember that the query we're using in `BooksPage` starts on a `Viewer`.

What we're doing here is saying, to use this component, it's concern is with something on a `Book` type. As long as we compose this `Book` fragment into the parent's fragment at a location where a `Book` fragment is valid then we'll be able to use this `BootItem` component.

The rule of thumb here is that if you are composing Relay components, make sure you compose their fragments.

If you get a warning that reads like the following:

`warning.js:36 Warning: RelayContainer: component` BookItem `was rendered with variables that differ from the variables used to fetch fragment` book.

This means that you forgot to include the child fragment in parent's query.

## Fragment Value Masking

There's another important feature of Relay that we haven't talked about yet: data masking.

The idea is **components can only see data they asked for explicitly**. If the component didn't ask for a specific field, Relay will actively hide that field from the component **even if that data was loaded**.

For instance, look at the queries on `BookItem` and `BooksPage` side-by-side . Notice that:

- `BookItem` loads the `slug` field for the book and
- `BooksPage` uses the `slug` field for the `Link` URL

For review, here's `BookItem`'s `Relay.QL` query:

relay/bookstore/client/src/components/BookItem.js

```
export default Relay.createContainer(BookItem, {
  fragments: {
    book: () => Relay.QL`
    fragment on Book {
      name
      slug
      tagline
      coverUrl
      pages
      description
      authors {
        count
      }
    }
    `,
  },
});
```

And `BooksPage`'s `Relay.QL` query:

relay/bookstore/client/src/components/BooksPage.js

```
export default Relay.createContainer(BooksPage, {
  initialVariables: {
    count: 100,
  },
  fragments: {
    viewer: () => Relay.QL`
    fragment on Viewer {
      books(first: $count) {
        count
        edges {
          node {
            slug
            ${BookItem.getFragment('book')}
          }
        }
      }
    }
    `,
  },
});
```



You might think that because we're calling `${BookItem.getFragment('book')}` in the `BooksPage` query that means that `slug` would be available to the `BooksPage` component, but **this is not the case**.

Fragment composition **does not** make that child-fragment's data available to our parent query. We must **explicitly** list the `slug` field in the `BooksPage` query if we want to be able to use that field in our `render` function of `BooksPage`.

Data masking is one of those features that can feel like an inconvenience at first, but it turns out to be super helpful as your app (and team) grows because it prevents bugs that result from changes outside of your component.

For instance, say that `BooksPage` was depending on the `slug` field from `BookItem` to be loaded. But, somewhere down the line, someone is cleaning up `BookItem` and they remove the `slug` field. What would happen? `BooksPage` would now break.

The idea is that **it is bad to have this coupling between components**. Each component should define everything it needs to render properly. Data masking is a way to ensure that every component explicitly defines the set of data it needs to operate.

### Masking Works Both Ways

This masking also goes the other way: since we **do** define that `BooksPage` (the parent) needs the `slug` field, even though we pass the book (via `<BookItem book={bookEdge.node}/>`) **the child cannot access `slug`** if it does not ask for it explicitly.

Any Relay manged props will have masking applied to them according to the queries. So be mindful of data masking when writing your applications – if you find that you unexpectedly have missing data (that you know you loaded somewhere else), check your queries to make sure you've explicitly request those specific fields in the component where they're needed.

## Improving the `AuthorPage`

Now that we can render a list of books, a nice touch to the author's page would be to show a list of the books that person has authored.

Given what we've just covered, we know that we can add this list of books by doing the following:

1. Request the author's `books` in our Relay query
2. Include the `BookItem` fragment in this query
3. Take the list of the author's books and render a `BookItem` for each one

Let's do this now.

## Requesting an Author's Books

Right now the `AuthorPage` only includes the `count` of the number of books an author appears on. Let's change the `AuthorPage` query to include more data on author's books in order to show a complete book thumbnail and link to them:

**relay/bookstore/client/src/components/AuthorPage.js**

```
71  export default Relay.createContainer(AuthorPage, {
72    fragments: {
73      author: () => Relay.QL`
74      fragment on Author {
75        _id
76        name
77        avatarUrl
78        bio
79        books(first: 100) {
80          count
81          edges {
82            node {
83              slug
84              ${BookItem.getFragment('book')}
85            }
86          }
87        }
88      }`,
89    },
90  });
```

This change shows one of the things that's great about Relay and GraphQL. Consider what this type of change might look like if we were working with a traditional REST-based API. We'd have to write code that looks up the author's `id` and makes a request to the server asking for that author's books. We'd have to add code to gracefully handle errors. Here, with Relay, all we have to do is add a `books()` section to the query and populate it with the fields we want.

Notice that we just hard-code that we want the first 100 books. In a real app, we'd probably set a variable like we did on the `BooksPage`.

The other thing we have to remember to do is include the `BookItem` fragment `book`. Again, note that the "parent" query here is `author`. You might recall that this is on an `Author` type up in the router. But, you don't really need to remember that. All you need to know is that `BookItem`'s `book` fragment is on type `Book` and so you can place it in your query wherever a `Book` type is allowed.



**How do I know what fragments my child components need?**

If you're building your first Relay app it might not be immediately clear how you even know what fragments to request. Essentially, it should be part of the documentation of the component. In the same way that in order to use a "regular" component, you're probably going to know what `props` it needs, in the same way, the author of the component you're using needs to document what fragments you'll use to render that component.

### `AuthorPage renderBook`

Now that we have the books we can render them like so:

**relay/bookstore/client/src/components/AuthorPage.js**

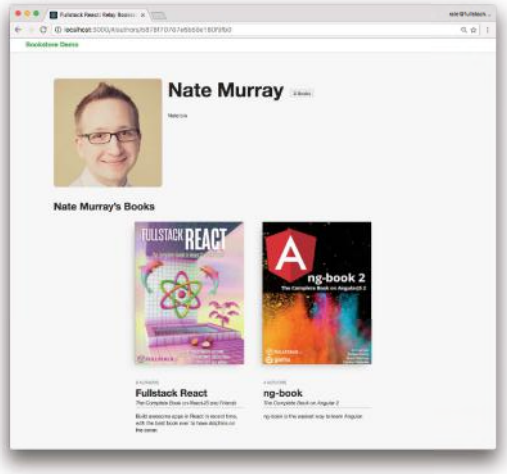
```
9   class AuthorPage extends React.Component {
10    renderBook(bookEdge) {
11      return (
12        <Link
13          to={`/books/${bookEdge.node.slug}`}
14          key={bookEdge.node.slug}
15          className='five wide column book'
16        >
17          <BookItem
18            book={bookEdge.node}
19          />
20        </Link>
21      );
22    }
```

and in the `render()` function:

```
render() {
  const author = this.props.author;
  const books = this.props.author.books.edges.map(this.renderBook);

  return (
    {/* ... truncated  */}

      <div className='sixteen wide column'>
        <h1>{ author.name }&rsquo;s Books</h1>
        <div className='ui grid centered'>
          { books }
```

```
        </div>
      </div>

    {/* ... truncated */}
  );
}
```

Here's a screenshot of what our author page looks like now that we have the books added in:




**Author with Books Page**

## Changing Data With Mutations

Now we've seen how to:

1. **read a single record** of data from Relay
2. **read a list** of data from Relay
3. load data from **child components**



There's one critical issue we haven't covered: **changing data**. In Relay, changing data is done through *mutations*.

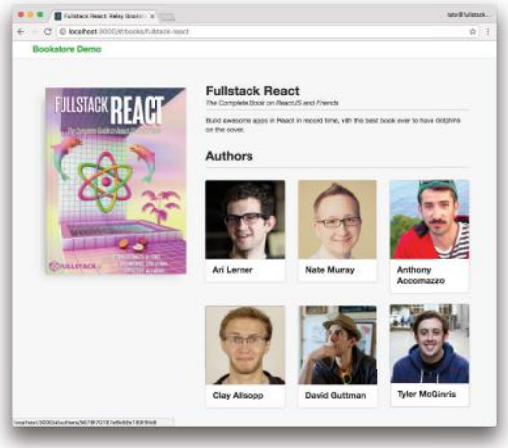
We're going to add the ability to edit the metadata of a book.

First, let's create a new page for an individual book. We'll read the initial data from Relay, as have been doing. Then we'll talk about how to change the data with mutations.

## Building a Book's Page

The read-only version of the individual book page doesn't have a lot of new ideas in it, so let's talk through it briefly.

First, here's a screenshot:




**Individual Book Page, Read Only**

You can see that we show the basic book image, each author, and the book cover.

Here's the Relay container:

**relay/bookstore/client/src/steps/BookPage.reads.js**

```
export default Relay.createContainer(BookPage, {
  fragments: {
    book: () => Relay.QL`
    fragment on Book {
      id
      name
      tagline
      coverUrl
      description
      pages
      authors(first: 100) {
        edges {
          node {
            _id
            name
            avatarUrl
            bio
          }
        }
      }
    }`,
  },
});
```

In this fragment we're loading the `Book` metadata as well as the first 100 authors.

Let's look at the `render()` function:

**relay/bookstore/client/src/steps/BookPage.reads.js**

```
  render() {
    const { book } = this.props;
    const authors = book.authors.edges.map(this.renderAuthor);
    return (
      <div className='bookPage sixteen wide column'>
        <div className='spacer row' />

        <div className='ui grid row'>
          <div className='six wide column'>
            <FancyBook book={book} />
          </div>
```



```

          <div className='ten wide column'>
            <div className='content ui form'>

              <h2>{book.name}</h2>

              <div className='tagline hr'>
                {book.tagline}
              </div>

              <div className='description'>
                <p>
                  {book.description}
                </p>
              </div>

            </div>

            <div className='ten wide column authorsSection'>
              <h2 className='hr'>Authors</h2>
              <div className='ui three column grid link cards'>
                {authors}
              </div>
            </div>

          </div>
        </div>

      </div>
    );
  }
```

Most of the markup here is `div`s with Semantic UI classes. Basically we're just show the book information with a reasonable layout.

For each author edge we call `renderAuthor`:

**relay/bookstore/client/src/steps/BookPage.reads.js**

```
12  renderAuthor(authorEdge) {
13    return (
14      <Link
15        key={authorEdge.node._id}
16        to={`/authors/${authorEdge.node._id}`}
17        className='column'
18      >
19        <div className='ui fluid card'>
20          <div className='image'>
21            <img src={authorEdge.node.avatarUrl}
22              alt={authorEdge.node.name}
23            />
24          </div>
25          <div className='content'>
26            <div className='header'>{authorEdge.node.name}</div>
27          </div>
28        </div>
29      </Link>
30    );
31  }
```

## Book Page Editing

Now let's add in some basic editing for this book's data. What we'd like to be able to do is click on any field like the title or description and edit it in-place. To make this simple, let's use the existing inline-edit component React Edit Inline Kit[167].

React Edit Inline Kit (or RIEK for short) has a simple API. We specify:

1. The original value
2. The property name of this value
3. A callback to run when this value changes

You can view the docs for RIEK here[168] and you can view the demos here[169]

[167] https://github.com/kaivi/riek
[168] https://github.com/kaivi/riek
[169] http://kaivi.github.io/riek/



So that we can isolate what we're doing, let's integrate RIEK into our app **without changing the data in Relay**. The first step is just to get the inline form editing working and then we'll deal with what to do with the changes once we have them.

Here's our new `render()` function on `BookPage` with `RIEK`:

**relay/bookstore/client/src/steps/BookPage.iek.js**

```
43  render() {
44    const { book } = this.props;
45    const authors = book.authors.edges.map(this.renderAuthor);
46    return (
47      <div className='bookPage sixteen wide column'>
48        <div className='spacer row' />
49
50        <div className='ui grid row'>
51          <div className='six wide column'>
52            <FancyBook book={book} />
53          </div>
54
55          <div className='ten wide column'>
56            <div className='content ui form'>
57
58              <h2>
59                <RIEInput
60                  value={book.name}
61                  propName={'name'}
62                  change={this.handleBookChange}
63                />
64              </h2>
65
66              <div className='tagline hr'>
67                <RIEInput
68                  value={book.tagline}
69                  propName={'tagline'}
70                  change={this.handleBookChange}
71                />
72              </div>
73
74              <div className='description'>
75                <p>
76                  <RIETextArea
77                    value={book.description}
78                    propName={'description'}
```

```
                    change={this.handleBookChange}
                  />
                </p>
              </div>

            </div>

            <div className='ten wide column authorsSection'>
              <h2 className='hr'>Authors</h2>
              <div className='ui three column grid link cards'>
                {authors}
              </div>
            </div>

          </div>
        </div>

      </div>
    );
  }
```

We've added three new components using `RIEInput` and `RIETextArea`. All three will call `handleBookChange` when the data changes. Here's the current implementation:

**relay/bookstore/client/src/steps/BookPage.iek.js**

```
  handleBookChange(newState) {
    console.log('bookChanged', newState, this.props.book);
  }
```

All we're going to do at this point is **log out the changes to `console`**.

Here's a screenshot of what happens when I change the tagline to `"The Complete Book on ReactJS and Dolpins"`:




**`console.log` the tagline**

You can see here that `newState` has the **new** values we need to update the `book`. Our task at hand is to **tell Relay about this change via a *mutation***.

Just to be clear, `RIEK` is a completely arbitrary choice for a forms library, selected here for convenience. It has nothing to do with Relay. The steps that we're about to take to create mutations in Relay could be done with any form library. If you'd like to build your own forms, that's perfectly fine. Checkout our chapter on Forms.

## Mutations

A *mutation* in Relay is an object that describes a change. Mutations are one of the most difficult aspects to get used to in Relay. That said, the complexity of mutations comes trade-offs that are inherent to building client-server applications. Let's take a look at the steps necessary to create mutations in Relay:

To mutate data in Relay:

1. We define a **mutation object**
2. We create an instance of that object, passing configuration variables and then
3. We send it to Relay using `Relay.Store.commitUpdate`

There are 5 major types of mutations in Relay:

- `FIELDS_CHANGE`
- `NODE_DELETE`
- `RANGE_ADD`
- `RANGE_DELETE`
- `REQUIRED_CHILDREN`

We are going to update the fields for an existing object, and so we will write a `FIELDS_CHANGE` mutation.

We're not going to cover every type of mutation in depth in this chapter.

If you want to view the official documentation that describes each of these mutations, checkout the Official Mutations Guide[170].

It's also worth noting that mutations can be a bit difficult to understand the first time. Making mutations easier to use is one of the major goals of Relay 2.

## Defining a Mutation Object

Mutations in Relay are objects. To define a new mutation we subclass `Relay.Mutation`. Then when we want to *execute* a mutation, we create an instance of this class, configure it with the appropriate properties, and give it to Relay (which handles communicating with the server and updating the Relay store).

When we create a `Relay.Mutation` subclass, we have to define 6 things that describe the behavior of the mutation:

1. What GraphQL method are we using for this mutation?
2. What variables will be used as the **input** to this mutation?
3. What fields does this mutation depend on in order to run properly?
4. What fields could change as a result of this mutation?
5. If everything goes smoothly, what's the expected outcome of this mutation?
6. How should Relay handle the actual response that comes back from the server?

We specify each of these things by **defining a function for each one**.

This might be more care than we usually take when we're using a fire-and-forget API, but remember, by specifically describing each of these steps in our mutation we get the benefits of Relay managing the bookkeeping for our data.

This is much easier if we walk through a concrete example, so lets make an "update" with `FIELDS_CHANGE`.

Open up `client/src/mutations/UpdateBookMutation.js`. Starting at the top let's look at `.getMutation`:

[170] https://facebook.github.io/relay/docs/guides-mutations.html#content



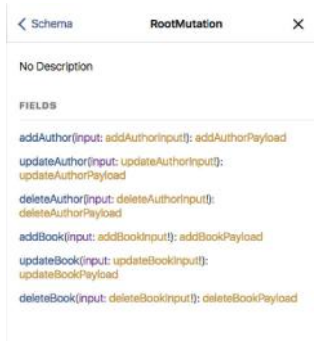
**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

```
5  export default class UpdateBookMutation extends Relay.Mutation {
6    getMutation() {
7      return Relay.QL`mutation { updateBook }`;
8    }
```

The first thing we need to specify when we create a `Relay.Mutation` subclass is the node of the GraphQL mutation. Here we've specified that we're using the `updateBook` mutation.

You can find this mutation in the schema by using GraphiQL and picking the `mutation` field at the root.

We can see here that `updateBook` takes a single argument `input`, which is of type `updateBookInput`, and it returns an item of type `updateBookPayload`.



updateBook Mutation

For `updateBook`, the values that we use for `updateBookInput` will be used as the **new values for the book** we specify.

That is, we'll send an `id`, which will be used to look up the specific `Book` we're changing, and we'll send new values such as the `name`, `tagline`, or `description`.

When it comes time to send this mutation, the values passed in to `updateBookInput` will come from the function `getVariables`:

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

```
10   getVariables() {
11     return {
12       id: this.props.id,
13       name: this.props.name,
14       tagline: this.props.tagline,
15       description: this.props.description,
16     };
17   }
```

This function describes how to create the arguments for `updateBookInput` – in this case we're going to look at `this.props` for values for the `id`, `name`, `tagline`, and `description`.

One of the things we have to be careful of is to ensure that this mutation has all of the fields it needs to operate properly. For instance, this mutation especially needs a `Book id` because that's how we're going to reference the object we're changing.

To do this, we specify `fragments`, much like we do for Relay container components:

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

```
19  static fragments = {
20    book: () => Relay.QL`
21      fragment on Book {
22        id
23        name
24        tagline
25        description
26      }
27    `,
28  }
```

Notice that in `getVariables` we're also sending along the `name`, `tagline`, and `description` – without checking if they have any value.

**Relay will mask `prop` values from mutations just like components**. So there's a subtle bug that could be introduced here. If we forget to specify the proper fragment, the mutation can accidentally set values to `null`.

What's the solution? In the same way that parent components need to include their child components' fragments, any component that uses a mutation also needs to include that mutation's fragments. We'll show how to do this when we use the `UpdateBookMutation` in our app below.

After we send this mutation it will be evaluated on the server. In this case, what changes on the server is pretty simple: we're updating the field values for **one object**.

That said, for many mutations the effects are probably more nuanced – we can't always know the full set of side effects that will occur from a mutation operation.

Furthermore, we don't actually have confirmation that this operation succeeded at all.

To deal with this, we're going to ask the server to send back to us all the fields that we think might have changed. To do this, we'll specify what's called a "fat query". It's "fat" because we're trying to capture everything that might have changed:



**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

```
30  getFatQuery() {
31    return Relay.QL`
32      fragment on updateBookPayload {
33        changedBook
34      }
35    `;
36  }
```

The fat query is a GraphQL query. In this case, we're just asking for the `changedBook`. So once the mutation is run on the server we're asking the server to send us back the new, updated values for the book that we (hopefully) changed.

Relay will take that `changedBook` (which is an object of type `Book`), look at its ID, and update the Relay Store accordingly.

However, before the server returns our *actual* response, we have the option to make a performance optimization and specify an "optimistic" response. The optimistic response answers the question: assuming this mutation executed successfully, what would be the response?

Here's our implementation of `getOptimisticResponse`:

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

```
49  getOptimisticResponse() {
50    const { book, id, name, tagline, description } = this.props;
51
52    const newBook = Object.assign({}, book, { id, name, tagline, description });
53
54    const optimisticResponse = {
55      changedBook: newBook,
56    };
57    console.log('optimisticResponse', optimisticResponse);
58    return optimisticResponse;
59  }
```

In this function we're merging together the old `book` with the argument values. Our optimistic response returns a `newBook` that looks like the response we're about to get from `getFatQuery`.

This mutation is straightforward: we have the original book and we have the updated fields. In this case, we know what the result of the mutation is going to be **without even asking the server**.

So what we can do is fake to the user that their operation was successful. This can give our user the feeling of a super-responsive app, because they're able to see the effects of their changes **immediately** without waiting for a network call.

If the mutation succeeds, the user is none the wiser.

If the mutation fails, then we've given the user false confirmation. But we'll still have the opportunity to handle that case and inform the user, perhaps telling them that they need to try again.

If the consistency trade-offs are acceptable to your application, optimistic responses can greatly improve the feel of the response time of your app.

When the real response is returned from the server, Relay needs to be told how to handle that data.

Because there are several different ways to mutate data, Relay mutations must implement a `getConfigs` method which describes the *way* Relay is going to handle the actual data that changed.

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

```
38  getConfigs() {
39    return [ {
40      type: 'FIELDS_CHANGE',
41      fieldIDs: {
42        changedBook: this.props.book.id,
43      },
44    } ];
45  }
```

In this case, because we're using a `FIELDS_CHANGE` mutation, we're specifying that we'll look at `changedBook` to find an ID that matches the `Book` we're talking about.

## Inline Editing

Now that we have our mutation defined, we're now prepared to implement click-to-edit on our `BookPage`.

Remember that our mutation is going to be making a query to check with the server for the current value of the `Books` data after the mutation finishes. Because of this, we need to **compose our mutation's fragment into our `BookPage` query**.

Just like a child Relay component needs its fragments composed, any mutations a component uses also needs to have its fragments composed.

Here is our `BookPage` query now that we're adding in the mutation's fragments:



**relay/bookstore/client/src/components/BookPage.js**

```
114 fragments: {
115   book: () => Relay.QL`
116   fragment on Book {
117     id
118     name
119     tagline
120     coverUrl
121     description
122     pages
123     authors(first: 100) {
124       edges {
125         node {
126           _id
127           name
128           avatarUrl
129           bio
130         }
131       }
132     }
133     ${UpdateBookMutation.getFragment('book')}
134   }`,
135 },
```

In the future, if you forget to add your mutation fragment to your query, you might get this:

```
1 Warning: RelayMutation: Expected prop `book` supplied to `UpdateBookMutation` to\
2  be data fetched by Relay. This is likely an error unless you are purposely pass\
3 ing in mock data that conforms to the shape of this mutation's fragment.
```

The solution: add your mutation fragment to the calling component's query.

Now we have everything in place to actually execute our mutation! Let's update `handleBookChange` to send out the mutation:

relay/bookstore/client/src/components/BookPage.js

```
  handleBookChange(newState) {
    console.log('bookChanged', newState, this.props.book);
    const book = Object.assign({}, this.props.book, newState);
    Relay.Store.commitUpdate(
      new UpdateBookMutation({
        id: book.id,
        name: book.name,
        tagline: book.tagline,
        description: book.description,
        book: this.props.book,
      })
    );
  }
```

Here we create a *new* object that is the merger of `this.props.book` and `newState` by using `Object.assign`. Next we execute our mutation by calling `Relay.Store.commitUpdate` and passing a new `UpdateBookMutation` to it.

Behind the scenes, Relay will:

- immediately update our view (and the Relay store) using the optimistic response
- call out to the GraphQL server and try to execute our mutation
- receive the reply from the GraphQL server and update the Relay store

Hopefully, at this point our optimistic response matches the actual response. But if not, the actual value will be represented on our page.

## Conclusion

Once we have the foundation in place, using Relay is a fantastic developer experience compared to hand-managing a traditional REST API.

Relay and GraphQL give a strong, typed structure to our data while giving us unmatched flexibility in changing our component data requirements.

Relay 2 is on the horizon and it promises to have even better performance, and a better developer experience (particularly around mutations). That said, Relay 1 is powerful enough to write apps with today.



## Where to Go From Here

If you want to learn more about Relay, here's a few resources:

- Learn Relay[171] – This is an introductory tutorial to Relay.
- `relay-starter-kit`[172] is a bare-bones implementation of Relay with a React app
- `relay-todomvc`[173] is an implementation of the popular TodoMVC app using Relay. If you're looking for more examples of mutations, this is a good place to start.
- `react-router-relay`[174] is the integration library we used in this chapter to use React Router with Relay
- A guide to authentication in GraphQL[175] - at some point you'll probably want to add authenticated pages to your server. Checkout this post for a good starting point
- Relay 2: simpler, faster, more predictable[176] – if you want to see where Relay is going, checkout these slides by Greg Hurrell

[171] https://www.learnrelay.org/
[172] https://github.com/relayjs/relay-starter-kit
[173] https://github.com/taion/relay-todomvc
[174] https://github.com/relay-tools/react-router-relay
[175] https://dev-blog.apollodata.com/a-guide-to-authentication-in-graphql-e002a4039d1#.z0vnf3846
[176] https://speakerdeck.com/wincent/relay-2-simpler-faster-more-predictable