# Hands-on Lab 5: GraphQL Server

**Estimated time: 60 minutes**

## Exercise 1:

## 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](https://github.com/rmosolgo/graphql-ruby)[[1]](#footnote-1), [Python](https://github.com/graphql-python/graphene)[[2]](#footnote-2), and [Scala](https://github.com/sangria-graphql/sangria)[[3]](#footnote-3). 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:

C:**\<**Your User>**\.**windows-build-tools**\p**ython27

Python comes with a script to add itself to your PATH.

To run that script in PowerShell:

1 > $env:UserProfile**\.**windows-build-tools**\p**ython27**\s**cripts**\w**in\_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:

1 > python

### Game Plan

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

* Create an [Express](http://expressjs.com/)[[4]](#footnote-4) 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:

$ 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](http://expressjs.com/)[[5]](#footnote-5).

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

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 }

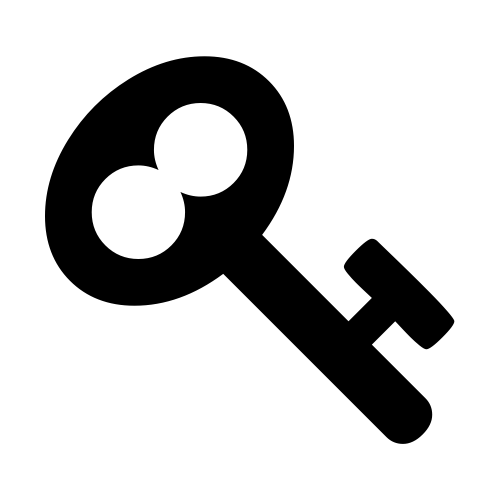
**$ curl -XPOST http://localhost:3000/graphql**

{"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](https://github.com/remy/nodemon#nodemon)[[6]](#footnote-6)to automatically restart your server when you make edits. npm install -g nodemon && nodemon index.js should do the trick.

### 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](https://github.com/graphql/graphql-js) [library1](https://github.com/graphql/graphql-js)23 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](https://github.com/graphql/express-graphql) [package](https://github.com/graphql/express-graphql)124 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:

**const** app = express();

**import** graphqlHTTP from 'express-graphql';

**import** { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';

**const** RootQuery = **new** GraphQLObjectType({

name: 'RootQuery',

description: 'The root query',

fields: {

viewer: {

type: GraphQLString,

resolve() {

**return** 'viewer!';

}

}

}

});

**const** Schema = **new** GraphQLSchema({

query: RootQuery

});

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/graphql -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:

**import** graphqlHTTP from 'express-graphql';

**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:

**const** RootQuery = **new** GraphQLObjectType({

name: 'RootQuery',

description: 'The root query',

fields: {

viewer: {

type: GraphQLString,

resolve() {

**return** 'viewer!';

}

}

}

});

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:

**const** Schema = **new** GraphQLSchema({

query: RootQuery

});

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 RootQuerynamed types.

Finally we hook it all up to Express:

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.

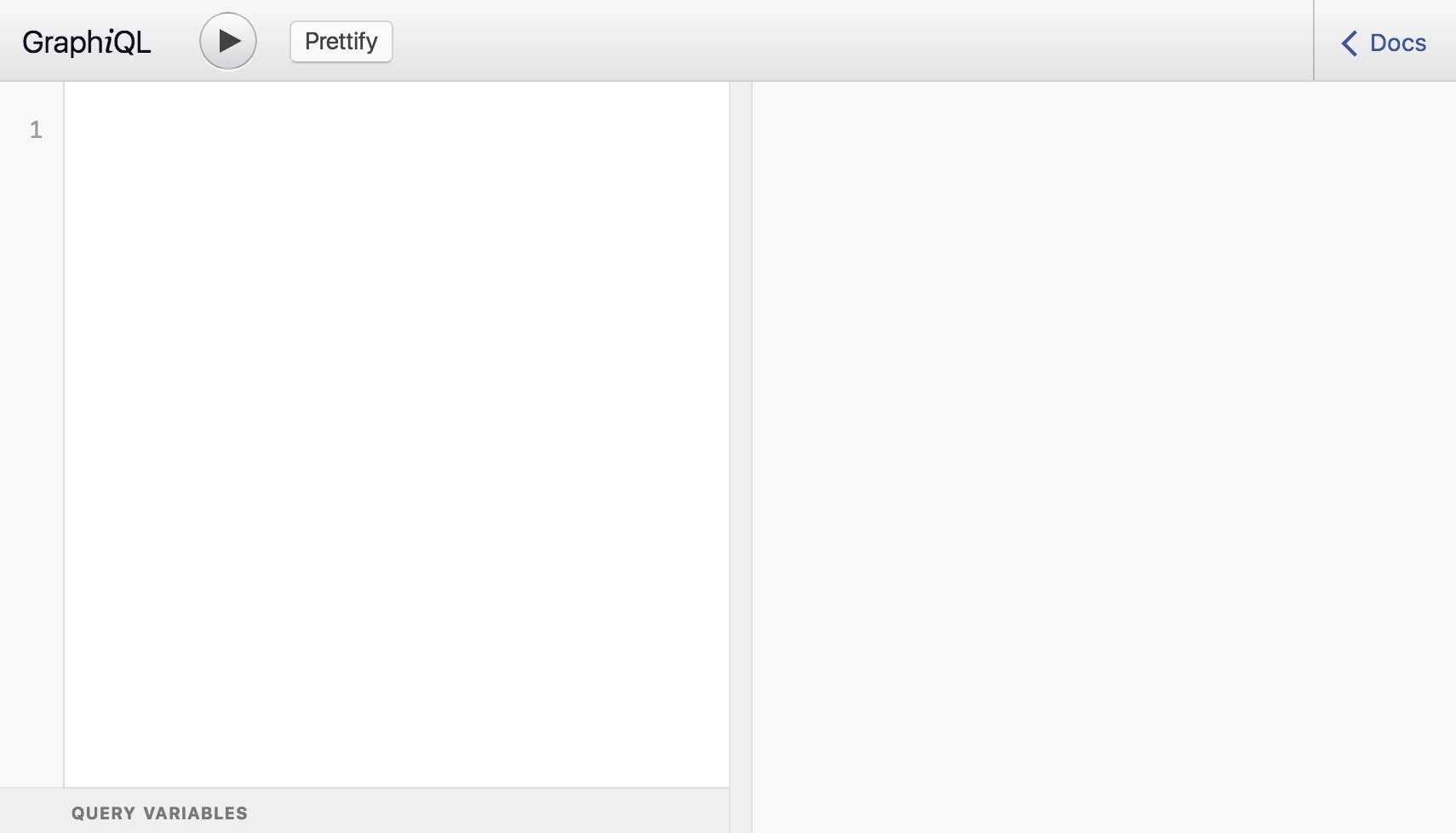
### Adding GraphiQL

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:

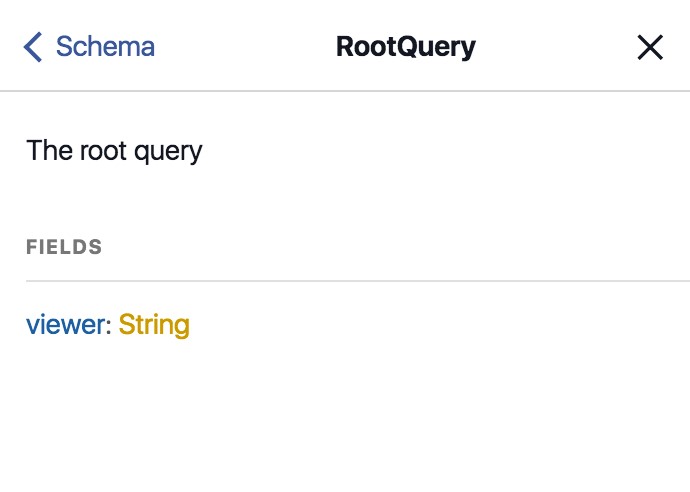
app.use('/graphql', graphqlHTTP({ schema: Schema, graphiql: **true** }));

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

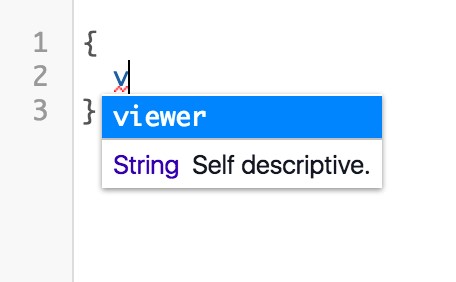


Empty GraphiQL

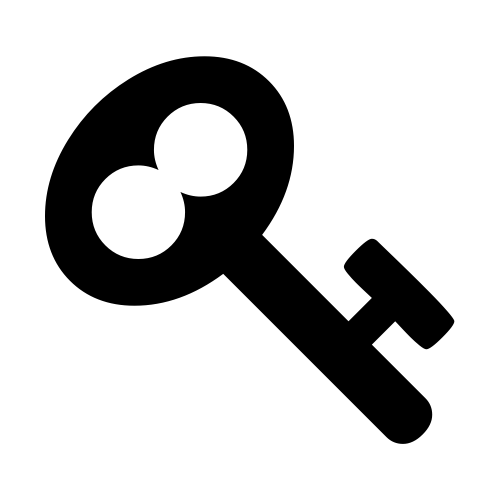
If you open the “Docs” sidebar, you’ll see all the information we entered about our Schema - the RootQuery, the description, and it’s viewer field:



Server Docs You’ll also get auto-complete for our fields:

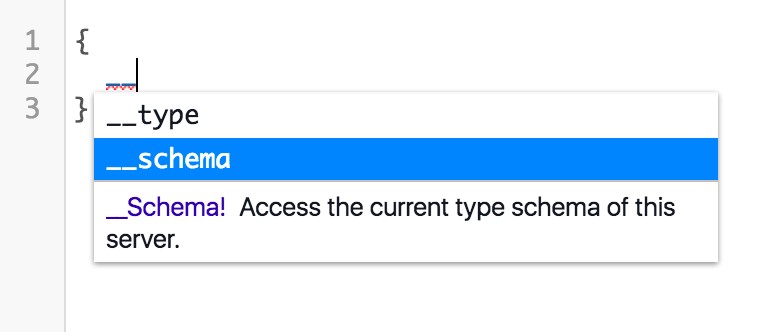


Server Autocomplete

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 documentatio[n](https://github.com/graphql/graphiql#getting-started)[[7]](#footnote-7) 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](https://facebook.github.io/graphql/#sec-Schema-Introspection)[[8]](#footnote-8). 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:

{

**"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:

**let** inMemoryStore = {};

**const** RootMutation = **new** GraphQLObjectType({

name: 'RootMutation',

description: 'The root mutation',

fields: {

setNode: {

type: GraphQLString,

args: {

id: {

type: **new** GraphQLNonNull(GraphQLID)

},

value: {

type: **new** GraphQLNonNull(GraphQLString),

}

},

resolve(source, args) {

inMemoryStore[args.key] = args.value;

**return** inMemoryStore[args.key];

}

}

}

});

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:

**const** Schema = **new** GraphQLSchema({

query: RootQuery,

mutation: RootMutation,

});

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

fields: {

viewer: {

type: GraphQLString,

resolve() {

**return** 'viewer!';

}

},

node: {

type: GraphQLString,

args: {

id: {

type: **new** GraphQLNonNull(GraphQLID)

}

},

resolve(source, args) {

**return** inMemoryStore[args.key];

}

}

}

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](https://github.com/brianc/node-sql)[[9]](#footnote-9) and [sqlite3](https://github.com/mapbox/node-sqlite3)[[10]](#footnote-10) 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'

}]

});

**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 sequence the promises 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](http://dbeaver.jkiss.org/)[[11]](#footnote-11), or you can verify that it has some data with this one-line command:

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

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:

{

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#Creating_a_Promise)[[12]](#footnote-12) to convert callback-based APIs to promises.

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.\_\_tableName);

**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)

}

}

}

});

**const** PostsConnectionType = **new** GraphQLObjectType({

name: 'PostsConnection',

fields: {

pageInfo: {

type: **new** GraphQLNonNull(PageInfoType)

},

edges: {

type: **new** GraphQLList(PostEdgeType)

}

}

}

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.

}

},

posts: {

type: PostsConnectionType,

args: {

after: {

type: GraphQLString

},

first: {

type: GraphQLInt

},

},

resolve(source, args) {

**return** loaders.getPostIdsForUser(source, args).then(({ rows, pageInfo

}) => {

**const** promises = rows.map((row) => {

**const** postNodeId = tables.dbIdToNodeId(row.id, row.\_\_tableName);

**return** loaders.getNodeById(postNodeId).then((node) => {

**const** edge = {

node,

cursor: row.\_\_cursor,

};

**return** edge;

});

});

**return** Promise.all(promises).then((edges) => {

**return** {

edges,

pageInfo

}

});

})

}

}

};

}

});

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](https://facebook.github.io/relay/graphql/connections.htm#sec-Pagination-algorithm)[[13]](#footnote-13). For brevity, we’re only going to support the after and first arguments.

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

**export const** getPostIdsForUser = (userSource, args) => {

**let** { after, first } = args;

**if** (!first) {

first = 2;

}

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:

**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 everyone 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](https://en.wikipedia.org/wiki/Basic_access_authentication)[[14]](#footnote-14). 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:

**import** express from 'express';

**import** basicAuth from 'basic-auth-connect';

**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.

app.use(basicAuth(**function**(user, pass) {

**return** pass === 'mypassword1';

}));

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:

$ curl -XPOST -H 'content-type:application/graphql' http://localhost:3000/graphql -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:

app.use('/graphql', graphqlHTTP((req) => {

**const** context = 'users:' + req.user;

**return** { schema: Schema, graphiql: **true**, context: context, pretty: **true** };

}));

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](https://facebook.github.io/relay/docs/api-reference-relay-mutation.html#getfiles)[[15]](#footnote-15)[[16]](#footnote-16) discusses how files are handled, and you can find [examples elsewhere1](http://stackoverflow.com/a/35585482)34 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 [Relay1](https://facebook.github.io/relay/)35, 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](https://github.com/graphql/graphql-relay-js)[[17]](#footnote-17) 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:

**import** {

connectionDefinitions

} from 'graphql-relay';

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

**const** resolveId = (source) => {

**return** tables.dbIdToNodeId(source.id, source.\_\_tableName);

};

**export const** UserType = **new** GraphQLObjectType({

name: 'User',

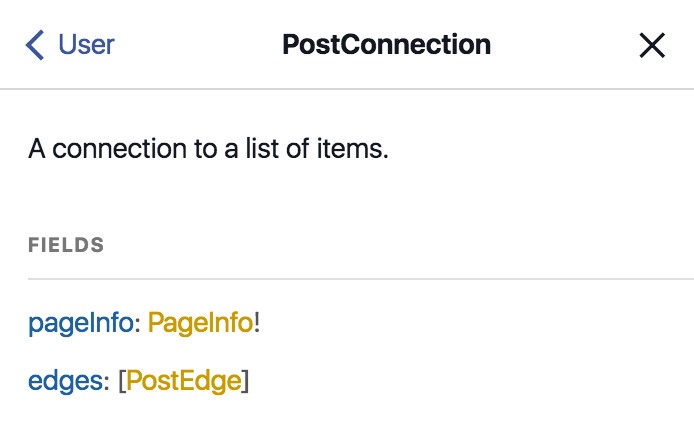
And at the very bottom, add this one-liner to define PostsConnectionType:

**const** { connectionType: PostsConnectionType } = connectionDefinitions({ nodeType\

: 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](https://github.com/graphql/graphql-relay-js#object-identification)[[18]](#footnote-18) and [working with Relay-compatible mutations](https://github.com/graphql/graphql-relay-js#mutations)[[19]](#footnote-19). 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:

{

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](https://secure.phabricator.com/book/phabcontrib/article/n_plus_one/)[[20]](#footnote-20) 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 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](https://github.com/facebook/dataloader)[[21]](#footnote-21) 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:

**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](https://github.com/facebook/dataloader#getting-started)[[22]](#footnote-22) and consider watching [this talk by its maintainer](https://www.youtube.com/watch?v=OQTnXNCDywA)[[23]](#footnote-23).

1. <https://github.com/rmosolgo/graphql-ruby> [↑](#footnote-ref-1)
2. <https://github.com/graphql-python/graphene> [↑](#footnote-ref-2)
3. <https://github.com/sangria-graphql/sangria> [↑](#footnote-ref-3)
4. [http://expressjs.com](http://expressjs.com/) [↑](#footnote-ref-4)
5. [http://expressjs.com](http://expressjs.com/) [↑](#footnote-ref-5)
6. <https://github.com/remy/nodemon#nodemon> [↑](#footnote-ref-6)
7. <https://github.com/graphql/graphiql#getting-started> [↑](#footnote-ref-7)
8. <https://facebook.github.io/graphql/#sec-Schema-Introspection> [↑](#footnote-ref-8)
9. <https://github.com/brianc/node-sql> [↑](#footnote-ref-9)
10. <https://github.com/mapbox/node-sqlite3> [↑](#footnote-ref-10)
11. <http://dbeaver.jkiss.org/> [↑](#footnote-ref-11)
12. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#Creating_a_Promise> [↑](#footnote-ref-12)
13. <https://facebook.github.io/relay/graphql/connections.htm#sec-Pagination-algorithm> [↑](#footnote-ref-13)
14. <https://en.wikipedia.org/wiki/Basic_access_authentication> [↑](#footnote-ref-14)
15. <https://facebook.github.io/relay/docs/api-reference-relay-mutation.html#getfiles>134<http://stackoverflow.com/a/35585482> [↑](#footnote-ref-15)
16. <https://facebook.github.io/relay/> [↑](#footnote-ref-16)
17. <https://github.com/graphql/graphql-relay-js> [↑](#footnote-ref-17)
18. <https://github.com/graphql/graphql-relay-js#object-identification> [↑](#footnote-ref-18)
19. <https://github.com/graphql/graphql-relay-js#mutations> [↑](#footnote-ref-19)
20. <https://secure.phabricator.com/book/phabcontrib/article/n_plus_one/> [↑](#footnote-ref-20)
21. <https://github.com/facebook/dataloader> [↑](#footnote-ref-21)
22. <https://github.com/facebook/dataloader#getting-started> [↑](#footnote-ref-22)
23. <https://www.youtube.com/watch?v=OQTnXNCDywA> [↑](#footnote-ref-23)