



GraphQL



OLLCO

GraphQL with Node.js and Apollo

Section 1: Introduction

Why GraphQL & Installing Node.js and Visual Studio Code

Most current applications likely uses some form of a REST API where we are making HTTP requests between the client and the server to send and receive data. Below is a visual representation of a REST API application where we have a client and a backend database/server.



The REST API would have multiple different endpoints/URLs that can be used to communicate with the server. For example: one endpoint for signing up, another for logging in, creating new data, reading all data and so on.

When we introduce GraphQL not a lot will change. We will continue to be able to use any client and any server but we will replace the REST API (with many endpoints) with a GraphQL API that has a single endpoint exposed.

GraphQL stands for the Graph Query Language and is something that operates over HTTP and therefore we can use any backend language and any database we want as well as any client (web, mobile, server) we want.



Why use GraphQL? GraphQL is fast, flexible, easy to use and simple to maintain. Below is a visual representation of a typical example of loading the necessary data for a page with a REST API and then with a GraphQL API.

We have a client and a server but the glue between the two is either a REST API or a GraphQL API. In both examples we would make a HTTP request to fetch the data necessary to render a page which displays a blog post i.e. the post details, comments made on the post and other blog posts made by that author.

Starting with the REST API we have a dozen or so endpoints for managing our application data. We would make HTTP requests to the various endpoints to get all the necessary data from the server. The server would determine what data to retrieve and send back to the client based on the endpoint requests made to the server.



The GraphQL API exposes a single endpoint which is a very important piece to the puzzle. This endpoint could be called anything we like (in the example below it is named `/graphql`). When making a HTTP request to this single endpoint we would also send along a GraphQL query.



A GraphQL query allows the client to describe exactly what data it needs from a server, the server gets all the data ready to send back to the client. This is the second most powerful piece to the puzzle. Instead of the server determining what data gets sent back, it is up to the client to request all of the data it needs. Therefore in the GraphQL case, it can request the post details, the post comments and other posts by that author all within a single GraphQL request.

The important difference between a REST API and a GraphQL API is that the latter allows the client to determine what data it gets back as apposed to the former HTTP endpoint which allows the server to determine what data comes back from and endpoint. Clearly 3 HTTP requests is more than a single HTTP request and therefore GraphQL is going to be more faster.

The greatest advantage of GraphQL is the flexibility. We could argue that for the REST API we could have had one endpoint that would return everything.



The above would be a perfectly fine approach and it would give the client everything it needs to render that page with just a single HTTP request. However, the problem with this solution is that we now have this one endpoint which is making way more database request than it was before.

It is getting big and slow i.e. from one database request to three database request in order to get all of the data necessary. For the desktop version of the application this may be perfectly fine as we may use the data right away. However, if we have a mobile version of our application to use the same backend. The problem with this is that the mobile application cannot change the data it gets back.



On mobile devices we have a whole set of considerations to take into account such as screen real estate, battery life, hardware and slow/expensive data. We want to make sure that we do not abuse the device else the users are going to get a poor user experience and unlikely to use the mobile application. This was the original reason for why GraphQL was created i.e. a flexible way for the individual clients to request exactly the data that it requires to use i.e. nothing more and nothing less. This is not an issue we would run into with the GraphQL API because it is more flexible.





The GraphQL API exposes a single endpoint for both desktop and mobile applications; however, the GraphQL query allows us to specify what data we would like to get back for example the desktop application would want everything while the mobile application would only want the post details.

The desktop query would require the server to do a lot more work i.e. make three database requests to get all of the data. However, for mobile the server does less work as the query requests less data i.e. one database request to get only the post details.

The REST API does not provide the same flexibility as we get the same response with both clients. With GraphQL it is for the individual client that determines what data it gets back from the server while with a REST API it is the server that determines what data it gets back to what endpoints.

Finally, with a REST API if the client requires different data this typically requires us to add a new endpoint or change an existing one. Using a GraphQL API, the client would just need to change its query making GraphQL API's much simpler to maintain compared to REST API.

To conclude, GraphQL creates fast and flexible APIs, giving clients complete control to ask for just the

data they need. This results in fewer HTTP requests, flexible data querying and in general less code to manage.

Before we can dive into looking at GraphQL we would need to install some applications/packages on our machines.

1) Node.js

We can visit <https://nodejs.org/en/> homepage to download node.js for our operating system. There are two versions: The LTS version (Long Term Support) and the Current release version. It is advisable to download the Current release, as this will contain all the latest features of Node.js. As at January 2020, the latest version is 13.6.0 Current.

We can test by running a simple terminal command on our machine to see if node.js was actually installed correctly:

```
:~$ node -v
```

If node.js was installed on the machine correctly the command would return the version of node that is running on the machine. If “command not found” is returned, this means node was not installed on the machine.

2) Visual Studio Code

A text editor is required so that we can start writing out our node.js scripts. There are a couple of open source code editors (i.e. free) that we can use such as Atom, Sublime, notepad++ and Visual Studio Code to name a few.

It is highly recommended to download and install Visual Studio Code as your code editor of choice as it contains many great features, themes and extensions to really customise the editor to fit your needs. We can also debug our node.js scripts inside of the editor which is also a great feature.

Links:

<https://code.visualstudio.com/>

<https://atom.io>

<https://www.sublimetext.com/>

Some useful extensions to install in Visual Studio Code are:

Babel ES6/7 (dzannotti), Beautify (HookyQR), Docker (Microsoft), Duplicate action (mrminc), GraphQL for VSCode (Kumar Harsh), npm (egamma), and npm Intellisense (Christian Kohler).

Section 2: GraphQL Basics: Schemas and Queries

What is a Graph?

When we talk about a 'graph' in GraphQL we are talking about a way to think about all of the application data and how that data relates to one another. Exploring what a graph is will provide a solid foundation for actually writing code in GraphQL.

We will use the example of a blogging application to explain what is a graph.



Types are things that we define when creating our GraphQL API, so in the above we have three types of User, Post and Comments.

When we define the types that make up our applications, we also define the fields associated with each type i.e. the individual data we want to store/track.

We also have relationships between the Types. In the above example, if a user creates a post that post is associated with the user. Therefore, the user can have many posts via the posts property. This also means each post belong to a user via the author property.

This should feel familiar because it is exactly how we would represent our data with any standard database applications whether SQL or NoSQL database (GraphQL does not care what backend database used to store the data).

To conclude a graph has types, fields and associations between various types and this is the mindset to have when thinking and representing data in GraphQL.

GraphQL Queries?

There are three major operations we can perform on a GraphQL API: query, mutation and subscription. Query allows us to fetch data, mutation allows us to change data and subscription allows us to watch data for changes.

We start the GraphQL syntax with the operation type. For a query operation this is “query” keyword in all lower case followed by an opening and closing curly brackets. Inside of the curly brackets we need to specify what fields we want from the GraphQL API. This will be a valid GraphQL query which will

fetch some data based on the query we specified from the server. Example query operation below:

```
query { hello }
```

The above GraphQL syntax looks very similar to a JSON or JavaScript object. If we were to execute the code on a GraphQL API which has this field we would receive a JSON data response like the below example:

```
{ "data": { "hello": "Hello World!" } }
```

We have an object with a single property “data” which has all of the data requested. In this example we requested a single field “hello” which has the data that came back from the server which is the string “Hello World!”.

We can request as many or as less data from the server as we need with a GraphQL query but typically it would be more than just one field. We would use a space or a new line to add multiple fields and would not need a delimiter such as a comma as we would see with JSON or JavaScript objects.

```
query { first_name last_name }
```

If we were to query a field name that the server does not understand we would get back a feedback error before even sending the request within the GraphQL playground to inform us that the field does not exist and it would provide suggestion to other fields which do exist.

All GraphQL APIs are self documenting which is not a feature we get with REST APIs. With REST APIs someone needs to manually write and update documentations. We do not run into this problem with GraphQL because our request dictates our response because we specify the exact things we want

access to and so we know how the response is going to look like. This is all possible because GraphQL API exposes an application schema which describes all of the operations that could be performed and the data we have access to when making requests to the GraphQL API.

GraphQL Nested Queries?

When we query on an object, we need to specify what fields on that object we would want to fetch. We cannot ask for everything because that defeats the purpose of GraphQL which requires the client to specify all the data it needs i.e. nothing more and nothing less.

To select fields from an object (Type) we use curly brackets after the object field name and within the curly brackets we would list the fields we want access to.

```
query { user { id name } }
```

When we query a custom Type (e.g. User!) this will return back an object with all of the fields we asked for i.e. in the above case this is the id and name fields:

```
{ "data": { "user": { "id": "123abc", "name": "John Doe" } } }
```

The structure of the data returned matches the structure of the query where “user” is on the top level and “id” and “name” are nested under “user” on the lower level.

We can query some data off of an array of objects of a custom Type (e.g. User![]) using the same syntax format as the above.

```
query { users { name } }
```

This will return an array of users object. For each user we are getting back the field we asked for i.e. the name field.

```
{ "data": { "users": [ { "name": "John Doe" }, { "name": "Sarah Browne" } ] } }
```

When looking at the Schema of a GraphQL API we would notice the syntax used which describe the fields available and the type of data we would get back from the field query, for example:

```
hello: String!
```

```
user: User!
```

```
users: [User!]!
```

The string to the left is the field we can query while the string on the right after the colon and space is the Type we would expect to get back. So in the above the field hello will return back a string data type while user will return a User object data type and users field will return an array of User object data type.

The exclamations mark indicates we would always get the data type specified before it and it cannot be null. If the exclamation mark was missing this would indicate that we may get back a null data from the field.

Useful link: <https://graphql-demo.mead.io/> — play around and write queries in the GraphQL playground.

Setting up Babel

Babel is a JavaScript compiler which allows us to write some code to babel and babel returns other code. This allows us to take advantage of cutting edge features in our code but still have code (output from Babel) which runs in a wide range of environments. Therefore, if we write an application which takes advantage of the arrow function but we want to run our code in older browsers, we are going to need a way to convert that arrow function to something that the older browser can understand. Babel will compile the code into code that is understood by older browsers. We are using Babel to access the ES6 import/export syntax from node. We can learn more about and play around with Babel on the following link: <https://babeljs.io/>

We will be using Visual Studio Code and the integrated terminal for Linux/Mac users (if using Windows, it is suggested to use a terminal emulator such as cmdr at <https://cmdr.net/>).

For setup, we would want to create a new directory such as graphql-basics and navigate to that directory within the terminal. We would then run the npm init command to create a package.json file for our project directory using the default values.

After creating the package.json file we would run the following command to install babel-cli and babel-preset-env packages. This will add the packages to our package.json file and add a node_modules directory within our project directory.

```
:~$ npm install babel-cli babel-preset-env
```


The babel-cli package will allow us to run a command to compile Babel while the babel-preset-env tells Babel exactly what it should change.

Once this has been installed we would create a new file in the root called .babelrc which will tell Babel to use the babel-preset-env preset so that our Import/Export code gets converted correctly and is compatible with node. The .babelrc is a JSON file whereby we set a presets property and set its value to an array where we provide as strings the names of all the presets we want to use, in this case we installed one which was the babel-preset-env (i.e. env). Babel is now all configured.

.babelrc:

```
{ "presets": [ "env" ] }
```

We need to create a src directory in the root of our project directory and within this directory we need a file as an entry point for our application which we can name as index.js. In the index.js we can add some content such as `console.log('Hello GraphQL!')` to make sure the file runs successfully.

Finally we need to update the package.json file to add a new script which we can run.

package.json:

```
{ ..., "scripts": { "start": "babel-node src/index.js" } }
```

We can now run the command in the terminal `npm run start` to run babel node in local development to compile and run our src/index.js file code. We should see "Hello GraphQL!" printed if successful.

ES6 Import/Export Syntax

The ES6 import/export syntax (like with require) is going to allow us to break up our application into multiple distinct files. Therefore, instead of having one huge file we can break our application code into many smaller files each concerned with some specific logical goal. This also allows us to load in code from other third party libraries we install. Below is an example of using the import/export syntax:

src/myModule.js:

```
const message = 'Hello from myModule.js';  
export { message };
```

The export statement allows us to export variables, functions, objects from a file. We do this by using the export keyword followed by curly brackets and within the brackets we list all of the things we want to export from the file.

src/index.js:

```
import { message } from './myModule';  
console.log(message);
```

The import statement allows us to import files and have access to the things that was exported from that file. The import statement has four major components, the first in the import keyword, the second is the list of all the things we want to import from the file which we wrap in the curly brackets, the third is the from keyword and fourth is the path to the import file. We can leave off the extension of the file.

The above example is known as a named export because the export has a name. We can export as many named exports we would like and the file importing the exports can select some or all of the exported things.

Aside from the name export there is another export we can use which is known as the default export. The difference between the two exports is that the default export has no name and you can only have one.

src/myModule.js:

```
const message = 'Hello from myModule.js';  
const name = 'John Doe';  
export { message, name as default };
```

The syntax is the same as a named export but we tag it as a default by using the `as default` keyword. This will select the one named export as the default export.

src/index.js:

```
import myName, { message } from './myModule';  
console.log(myName);
```

To grab the default export we give it a name after the import keyword. We use a comma if we are also providing a named export. This will grab all export types i.e. default and named exports. The import name does not match the export name (and does not need to) because a default export has no name. We are importing by it's role of the default which is why we are getting the correct import.

Creating Your Own GraphQL API

Using GraphQL Playground is a temporary way to make sure our server is working as expected when we perform various operations on it. In a real world scenario the queries will come from a mobile device, another server or a web browser. The GraphQL Playground is a great starting point for learning the various operations available to us.

It is important to note that the Graph Query Language is just a specification for how it should work e.g. how should queries work. At the end of the day it is not an implementation and therefore it is up to the individual developers to take the documents that describes GraphQL and actually implement it for that environment, whether Android, IOS, JavaScript, Node, Python, C#, etc. We need implementations of GraphQL in those environments. Therefore, we would need to find an implementation of the GraphQL spec that works with Node.js (as we are using Node.js as our backend language in this guide). This is similar to JavaScript EcmaScript specification describing how JavaScript should work and then we have various implementation of the JavaScript spec e.g. Chrome and Node use the V8 engine, Mozilla using Spidermonkey and Microsoft using Chakra.

We can go to the link to view the GraphQL spec: <https://spec.graphql.org/> — This document will describe/explain how GraphQL works. It is for the developer of the other languages for the implementation. This means we might have multiple options to pick from when we are using GraphQL in a specific environment. The tool we are going to be using to get GraphQL running on Node.js is GraphQL-Yoga which we can install by running the command in the terminal within our project directory.

```
:~$ npm install graphql-yoga
```

The documentation for graphql-yoga tool can be found on <https://github.com/prisma-labs/graphql-yoga>.

The reason for using this tools is because it provides the most advanced feature set allowing us to tap into everything that GraphQL has to offer and it also comes with a very easy setup process so that we can get up and running very quickly. We can use the same tools to build very advanced API with all the features we would expect.

Below is the code to setup our own GraphQL API.

src/index.js:

```
import { GraphQLServer } from 'graphql-yoga';
const typeDefs = `type Query { hello: String! }`
const resolvers = { Query: {
  hello( ) { return 'This is my first query!'; }
} };
const server = new GraphQLServer( { typeDefs: typeDefs, resolvers: resolvers } );
server.start( ( ) => { console.log('The server is up and running') } );
```

We import GraphQLServer named export from graphql-yogo which will provide us the tool to create a new server. Since the imports is from a npm library, we only need to reference the library name. This is all we need to import to get GraphQL up and running.

There are two things we need to define before we can start our server, the first is our Type definitions and the second are the resolvers for the API.

Type definitions is also know as the application schema. The application schema is important because it defines all of the operations that can be performed on the API and what are our custom data types look like. Resolvers are nothing more than functions that run for each of the operations that can be performed on our API.

The type “Query” which must be in all uppercase because it is one of the three builtin types. Within the curly brackets we define all of the queries we want to support (in the above example we define a single query). To define the query we start with the query name followed by a colon. We then define the type that should come back when the query is executed. This is a valid Type definition for our API. If we do not want a null to be a returned value we must use the exclamation mark at the end of the type definition e.g. String!

The resolver object typically resembles what is defined in the type definition. The Query is a property which holds various methods, one method for each query defined. In the above example there is only one Query and therefore there is only one method.

To start up the server and query for that data we use two lines of code. The first is to declare a new server. The GraphQLServer takes in an object as its arguments where we define two properties i.e. the typeDefs and resolvers. Note since the values match the property name we could use the ES6 shorthand syntax of just the property name (e.g. typeDefs: typeDefs can be written as typeDefs).

Finally, we use the start method on the server to start everything up. We pass to it a callback function

to return a string to confirm the server is running. When we run `npm run start` to run our index.js code, the graphql-yoga by default will start the server/application on `localhost:4000`.

We should now see a brand new instance of GraphQL Playground for the GraphQL API we have setup. We can now query for hello which is the only thing that exists in our API and should see something like the below screenshot:



We now have our very first and simple GraphQL API server up and running.

Important note: any changes we make to our code i.e. adding new queries, we would need to shut down the server using control + c on your keyboard and restart the server using the `npm run start` command and refresh the page to see the changes.

Live Reload for GraphQL-Yoga

We can set things up to automatically restart the server (rather than manually) in the background anytime we make any changes to our application code using a library called nodemon.

We can run the following command in the terminal to install the library within our project directory:

```
:~$ npm install nodemon --save-dev
```

After installing the package as a dev dependency we would update the “start” script to use nodemon:

package.json:

```
{ ..., "scripts": { "start": "nodemon src/index.js --exec babel-node" } }
```

GraphQL Scalar Types

GraphQL comes by default with five main scalar types that we can use for our Type Definitions: String, Boolean, Int, Float and ID. Integers holds full numbers while float holds decimal numbers. The ID type is unique to GraphQL and is used to represent unique identifiers as strings.

A scalar value is a single discrete value and so a scalar type is a type that stores a single value. This is the opposite of a non-scalar type such as an object or an array which is a collection of discrete values. So with an object we have many properties while with an array we can have many elements in that array.

If we define a type definition with an exclamation mark, this states the data query must return that type and cannot return null.

Creating Custom Types

In GraphQL you are not limited to the five scalar types. We are able to create our own custom types which are essentially an objects i.e. a set of fields. To create a custom type it starts within the type definition. We use the type keyword followed by a type name of our choosing.

src/index.js:

```
const typeDefs = `
  type Query { me: User! }
  type User { id: ID! name: String! email: String! }
`

const resolver = { Query: { me() { return{ id: '012345', name: 'Beatrice', email: 'b.email.com' } } } }
```

When choosing a type name it is common practice to use a uppercase letter. The name also resembles the object we are modelling such as a User, Post, Product, etc.

Within the curly brackets we define all of the fields that make up the custom Type. This will look similar to defining Query definitions.

We would create a field name in the Query definition and set its value to the custom Type (previously we would set it to scalar types but are now able to use our custom Types).

Finally, we would setup the resolver function for the Query definition which returns data (an object) matching with the custom query type. We now have a query that returns a custom type.

Important Note: the resolver is returning static data back to the query and later on we will learn how to return dynamic data from a database.

In the above example we now have a query definition of me: User! that we can now query on. It is important to note when we are querying data in GraphQL, we have to be specific down to the individual scalar values and there querying the field “me” alone is not enough (unless it returned a scalar value). This is because the “me” field is returning an object which is not a scalar value. The individual fields within the “me” object are scalar values such as id, name, email and age. Therefore, our query would look something of the below screenshot in the GraphQL Playground:



The screenshot shows the GraphQL Playground interface. At the top, there's a tab labeled 'me' with a close button. Below it, there are buttons for 'PRETTIFY', 'HISTORY', and a URL bar showing 'http://localhost:4000/'. To the right of the URL bar is a 'COPY CURL' button. The main area is split into two panels. The left panel contains a query editor with the following code:

```
1 query {  
2   me {  
3     id  
4     name  
5     email  
6     age  
7   }  
8 }
```

The right panel shows the JSON response after clicking the play button (a large circle with a right-pointing triangle). The response is:

```
{  
  "data": {  
    "me": {  
      "id": "0123456",  
      "name": "John Doe",  
      "email": "j.doe@email.com",  
      "age": null  
    }  
  }  
}
```

On the far right, there are two vertical buttons: 'DOCS' and 'SCHEMA'.

Therefore, if we have a non scalar value, we have to provide a selection set for it using the curly brackets and selecting the actual scalar values want returning. We do not necessarily have to return all the values from the “me” object and can be selective on what we want returning back from our query statement.

We can now appreciate and see how we can model our application in terms of the custom types but there are a lot more questions we need to answer first for example how can we work with lists/array, how do we set relations between these custom types, etc.

Operation Arguments

GraphQL operation arguments allows us to pass data from the client to the server. So far all the data has been flowing in the opposite direction i.e. from the server to the client. There are many reason for why we would want to get data from our client to the server for example a signup form on the front end where we need to get this information through to the server so that it can save a new record to the database.

To create an operation argument we start with creating a Query definition.

src/index.js:

```
const typeDefs = ` type Query { greeting(name: String): String! } `
const resolver = { Query: { greeting(parent, args, ctx, info) {
  if(args.name) { return `Hello, ${args.name}!` }
  return 'Hello!';
} } };
```

The query will return a scalar type at the end of the day; however, we can determine what arguments this query should accept by adding round brackets after the query name and before the colon. This is where we define/list all of the arguments that could be passed along with the query and define whether the arguments are required or optional for the query.

We would give a name for the argument and then after the colon we need to be explicit about what type we are expecting. To make an argument required we would add an exclamation mark after the type definition of the argument (leaving this off would make the argument optional).

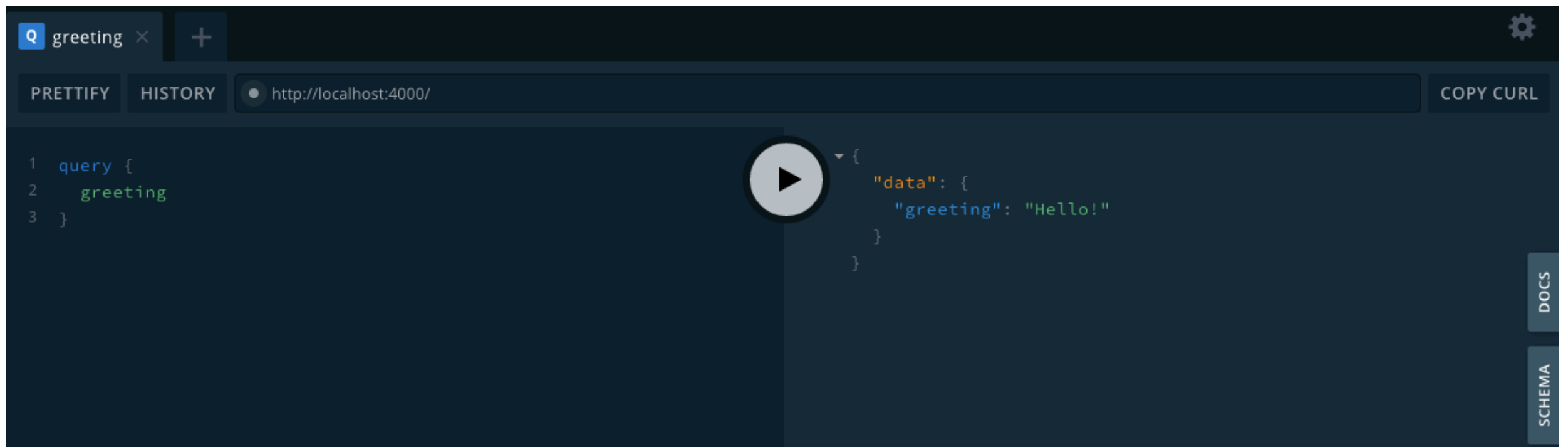
We would then setup the resolver method for this query. It is up to the resolver method to use the data to make some sort of dynamic response. There are four arguments in the specific order that get passed to all resolver functions and they are as follows:

- ◆ parent - this is a very common argument and is useful for working with relational data
- ◆ args - this contains the operation arguments supplied
- ◆ ctx - shortened for context and is useful for contextual data (e.g. if a user is logged in, context may contain the id of that user so that we can access it throughout the application)
- ◆ info - this contains information about the actual operation that were sent along to the server

When we want to make a query to the GraphQL API where a query has operational arguments setup, we would pass all of the arguments within the round brackets after the query field name. We specify the argument key we are setting a value for followed by a colon and then the value for that argument. This will allow us to successfully pass argument values from the client to the server:



If we were to query for greeting and leave the arguments off or use an empty string this would return only the greeting as seen in the below examples:



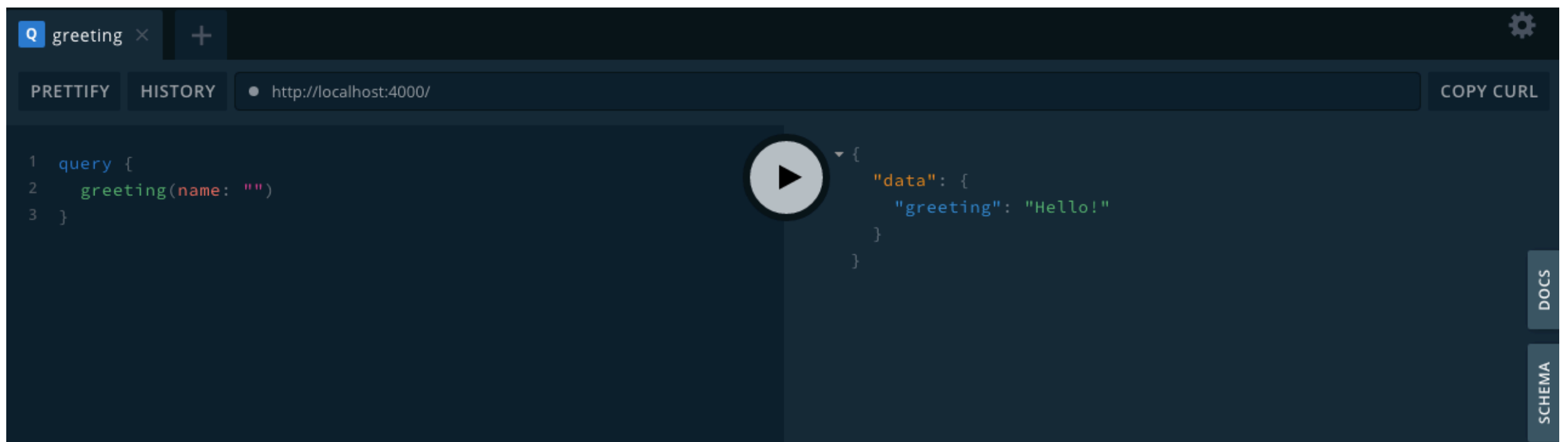
The screenshot shows the GraphQL Playground interface. The top bar includes a tab labeled 'greeting', a 'PRETTIFY' button, a 'HISTORY' button, a URL input field containing 'http://localhost:4000/', and a 'COPY CURL' button. The query editor on the left contains the following query:

```
1 query {  
2   greeting  
3 }
```

A play button is located between the query and the response. The response on the right is a JSON object:

```
{  
  "data": {  
    "greeting": "Hello!"  
  }  
}
```

On the right side of the interface, there are vertical buttons for 'DOCS' and 'SCHEMA'.



The screenshot shows the GraphQL Playground interface with the same top bar as the previous example. The query editor on the left contains the following query:

```
1 query {  
2   greeting(name: "")  
3 }
```

A play button is located between the query and the response. The response on the right is a JSON object:

```
{  
  "data": {  
    "greeting": "Hello!"  
  }  
}
```

On the right side of the interface, there are vertical buttons for 'DOCS' and 'SCHEMA'.

This is how we can pass data from the client over to the server using operation arguments. There are also no limitation to the number of arguments we can setup for a query.

Working with Arrays

We can send arrays back and forth between the client and the server. We can work with arrays of the scalar types as well as arrays of custom types. To create an array type we use square brackets, much like we would if we were to create arrays in JavaScript itself. Within the square brackets we specify the scalar type for all of the elements returned in the array.

src/index.js:

```
const typeDefs = ` type Query { grades: [Int!]! } `
const resolvers = { Query: { grades(parent, args, ctx, info) { return [89, 60, 78 ] } } };
```

Having an exclamation mark at the end of the square brackets will declare that our query would always return an array even if it is an empty array. We can also decide to use or not to use the exclamation mark on the internal scalar type. If omitted, this would mean that we can have null values within the array values. Note in the above we can still have an empty array even if we use the exclamation mark in the internal scalar type.

It is good practice to get into the habit of passing in all of the arguments to all of the resolver functions even if we are not going to use them.

We now have an array field setup and can now query our array in GraphQL Playground as seen in the below example:



Notice that we did not provide anything after the field itself when querying i.e. we did not provide a selection set using the curly brackets to specify what we want from the grades array. This is not something we can do for an array of scalar types such as an array of strings, array of integers, array of floats, etc. Specifying a selection set is only something we would do for an array of custom types.

In the above example we are sending an array from the server to the client. We can also send an array in the other direction as seen in the example below:

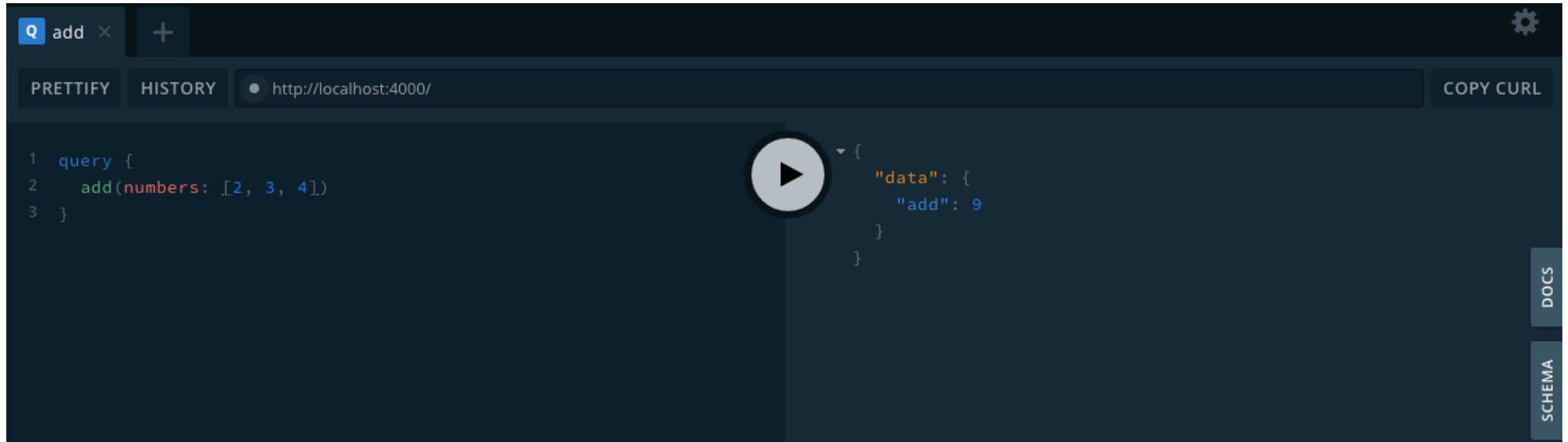
[src/index.js:](#)

```
const typeDefs = ` type Query { add(numbers: [Float!]!): Float! } `
const resolvers = { Query: { add(parent, args, ctx, info) {
  if(args.numbers.length === 0) { return 0; }
  return args.number.reduce( (accumulator, currentValue) => {
    return accumulator + currentValue;
  })
}
```

```
} );
```

```
} } };
```

The reduce is an array function which reduces an array to a single value. This function takes in a callback function and the callback function receives two arguments, the accumulator and currentValue. When the function iterates for the first time, the accumulator is the first element in the array and the currentValue is the second element in the array. On the next iteration the accumulator becomes the value of whatever the outcome was of the callback function and the currentValue becomes the next element within the array. This loops through all elements in the array until the array reduces down to a single value. In the above case we are adding all values within the array to return a single sum value from the array.



As mentioned above we can also create an array of custom types, below is an example of this:

src/index.js:

```
const users = [ { id: '1', name: 'Alice', email: 'alice@email.com', age: 24 }, { id: '2', name: 'Barry',  
                email: 'barry@email.com' }, { id: '3', name: 'Carl', email: 'carl@email.com', age: 54 } ];  
  
const typeDefs = `  
  type Query { me: User! users: [User!]! }  
  type User { id: ID! name: String! email: String! age: Int }  
`  
  
const resolvers = { Query: {  
  me(parent, args, ctx, info) { return { id: '012345', name: 'Beatrice', email: 'b.email.com' } }  
  users(parent, args, ctx, info) { return users; }  
} };
```

Each user within the users array matches the Users Type definition and we can therefore return the users array variable in the resolver function. We can now query for the users field in GraphQL Playground; however, as mentioned above we must always provide a selection set using the curly brackets to specify the field(s) within the custom type array. This is because the custom type array does not return scalar types and we must always specify the scalar types to return back from our query.

Below screenshot provides an example of a querying on a custom type array where we must provide a selection set for the fields we want returning back for each custom type element within the array:



We will notice that where an age value does not exist it simply returns null values because age is completely optional in our User Type definition (i.e. no exclamation mark after the Type definition for age: Int = nullable field).

Typically, when working with arrays of a custom types, we would want to provide other functionality to the client for example sorting or filtering the data. This will modify either the order of the items that come back and/or the number of items that come back altogether from the query.

We can achieve this by using operation arguments and array data types. Below is an example code:

src/index.js:

```
const users = [ { ... }, { ... }, ... ];
const typeDefs = `
  type Query { me: User! users(query: String): [User!]! }
  type User { id: ID! name: String! email: String! age: Int }
`
const resolvers = { Query: {
  me(parent, args, ctx, info) { return{ id: '012345', name: 'Beatrice', email: 'b.email.com' } }
  users(parent, args, ctx, info) {
    if (!args.query) { return users; } return users.filter((user) => {
      return user.name.toLowerCase( ).includes(args.query.toLowerCase( ));
    } );
  }
}
};
```

The filter array method iterates over each elements in an array and creates a new array of items for all truthy values returned from the callback functions (i.e. filtering out all falsey values from the array). The .toLowerCase method makes the callback function case insensitive while the .includes method checks if the string value contains another string value. We now have a way to filter the users by their name when making a query to the Users custom type.

Below is a screenshot of the GraphQL Playground and making a query to the Users custom type and providing a name filter query to reduce the number of Users data sent back:



Note: this is a case insensitive search and will return all users which has the letter 'R' in their name (in the above example only two records was returned back from the query). We can also omit the query entirely which will return all users data.

This concludes working with arrays with both scalar type and custom type queries.

Relational Data: Basics

Relational data in GraphQL is going to allow us to setup relationships between our types. For example, every post is written by some user and a user can have a collection of posts that they have actually worked on. What we would need to do is setup some relationship between the two types. This is going to allow us to perform some more advanced and real world queries.

To setup a relationship we first need to change the type definition for our custom types. If we want to our query types to have a relationship with another type, it has to be setup explicitly in the Type definition. Below is an example of creating a relationship between Post and User using an author relationship field.

src/index.js:

```
const users = [ { id: '1', ... }, { id: '2', ... }, ... ];
const posts = [ { ..., author: '1' }, { ... author: '1' }, ... ];
const typeDefs = `
  type Query {
    users(query: String!): [User!]!
    posts(query: String!): [Post!]!
  }
  type Post { id: ID! title: String! body: String! published: Boolean! author: User! }
  type User { id: ID! name: String! email: String! age: Int }
`
```


The data itself now requires to have a field which can be used to link the Post to the User. Therefore, in the posts array each item should have an author property with a value that links to the user id. This will provide enough data to make an association.

When setting up a field where the value is another custom type we would have to define a function that tells GraphQL how to get in the above case the author if we have the Post. We do this by defining a new root property on resolvers to go alongside of Query. This name should match our custom type definition name.

src/index.js:

```
...
const resolver = {
  Query: { posts(...) { return... }, ... },
  Post: { author(parent, args, ctx, info) { return users.find((user) => {
    return user.id === parent.author; } );
  } }
}
```

This property value is an object that holds methods for each of the fields that actually link to another type. This is a resolver method which has the same set of arguments of parent, args, ctx and info. The goal is to return the correct relational data i.e. in the above this is the author for the post. The post information lives on the parent argument and so we can use that to figure out how to join the relationship.

The first thing GraphQL will do when we run the Post query, it is going to run the Query: posts resolver method. GraphQL is going to see what data was requested and if we only had scalar types, this is where the query would end. However, in the above case we have also asked for author field and this does not live on the post data. Therefore, GraphQL for each individual post is going to call on the Post: author resolver method with the post object as the parent argument.

On the parent argument we can access things such as the post id, title, body, published or author properties. We can use the author property to iterate over the users array to find the correct user and return it. The find method is similar to filter method. The find method's callback function gets called one time for each element in an item until it finds a match (i.e. truthy value) giving back an individual object.

If we now go into the GraphQL Playground and query posts and want the author, we would also need to provide a selection set because the author returns a custom type and not a scalar. The query example below provides the author name for every posts:



The screenshot shows the GraphQL Playground interface. The query editor on the left contains the following query:

```
1 query {  
2   posts {  
3     id  
4     title  
5     body  
6     published  
7     author {  
8       name  
9     }  
10  }  
11 }
```

The response editor on the right shows the JSON output:

```
{  
  "data": {  
    "posts": [  
      {  
        "id": "10",  
        "title": "GraphQL 101",  
        "body": "Introduction to GraphQL...",  
        "published": true,  
        "author": {  
          "name": "Alice"  
        }  
      }  
    ]  
  }  
}
```

The interface includes a 'Q' icon, a 'posts' tab, a 'PRETTIFY' button, a 'HISTORY' button, a URL bar showing 'http://localhost:4000/', a 'COPY CURL' button, and a 'PLAY' button (a circle with a right-pointing triangle). On the right side, there are 'DOCS' and 'SCHEMA' tabs.

This is how we can setup a relationship in GraphQL. This relationship flows in a single direction i.e. the Post Custom Type has a link to the User Custom Type via the author property but the User Custom Type does not have a link to Post Custom Type.

Relational Data: Arrays

We will now look at how to create a relationship between the User Custom Type and the Post Custom Type for the other relationship direction. To do this we would create the relationship via a posts filed on the User Custom Type.

[src/index.js:](#)

```
const users = [ { id: '1', ... }, { id: '2', ... }, ... ];
const posts = [ { ..., author: '1' }, { ... author: '1' }, ... ];
const typeDefs = `
  type Query {
    users(query: String!): [User!]!
    posts(query: String!): [Post!]!
  }
  type Post { id: ID! title: String! body: String! published: Boolean! author: User! }
  type User { id: ID! name: String! email: String! age: Int posts:[Post!]! }
`
```

We now need to teach GraphQL how to get the posts for a user when it has the user. This is similar to how we taught GraphQL how to get the author when it has the post. What we need to setup now is

the User: post resolver method. It is important to remember that if one of our fields is not a scalar type we would have to setup a custom resolver function to teach GraphQL how to get the correct data.

src/index.js:

```
...  
const resolvers = {  
  Query: { users(...){ return... }, ... },  
  Post: { author( ... ) { return ... } },  
  User: { posts(parent, args, ctx, info) {  
    return posts.filter((post) => { return post.author === parent.id; } );  
  } };  
}
```

When the query runs, GraphQL is going to run the resolver methods for our Query: users. When GraphQL returns a value it is going to check if we requested any relational types i.e. posts and if so it will call the User: posts resolver method for every single user it has found. Each time the resolver method is called it will have a different parent argument value. We can therefore use the filter method on posts to find the posts where the author property matches with the parent.id value.

We can now go to GraphQL Playground and run a valid query for the users post field to return back the array of posts belonging to each user as seen in the below screenshot example:

Q users x +

PRETTIFY HISTORY http://localhost:4000/ COPY CURL

```
1 query {
2   users {
3     id
4     name
5     email
6     age
7     posts {
8       id
9       title
10    }
11  }
12 }
```

▶

```
{
  "data": {
    "users": [
      {
        "id": "1",
        "name": "Alice",
        "email": "alice@email.com",
        "age": 24,
        "posts": [
          {
            "id": "10",
            "title": "GraphQL 101"
          },
          {
            "id": "11",
            "title": "GraphQL 201"
          }
        ]
      }
    ]
  }
}
```

DOCS

SCHEMA

We will notice the value of the posts field is an array of the posts object containing the id and title. We now have a relationship setup in both directions as we saw in the diagram on page 10 (see below):



Section 2: GraphQL Basics: Mutations

What is a Mutation?

The GraphQL Query operator allows us to fetch/read data. Mutation is another GraphQL operation which allows us to create, update and delete data. This is the core feature of GraphQL operators required to perform CRUD operations on data stored on a server.

Creating Data with Mutations?

The Type definitions is where we would define all of the mutations we would support similar to how we have defined all of our queries we would want to support on our GraphQL API. We would setup a new Type definition for another built in type called Mutation. Within the curly brackets we would define all of the mutations we want our server to be able to perform. Defining a mutation is exactly the same as defining a Query and so we can reuse all of the knowledge/syntax we learned in Section 1. Below is an example:

src/index.js:

```
import uuidv4 from 'uuid/v4';  
const typeDefs = `  
  type Mutation { createUser(name: String!, email: String!, age: Int): User! }  
`
```

```

const resolvers = {
  Query: { ... },
  Mutation: {
    createUser(parent, args, ctx, info) {
      const emailTaken = users.some((user) => user.email === args.email);
      if (emailTaken) { throw new Error('Email taken.')}
      const user = { id: uuidv4(), name: args.name, email: args.email, age: args.age };
      users.push(user);
      return user;
    }
  }
};

```

In this example we have a mutation that allows us to create a new user. This takes in three parameters, the name, email and an optional age field. The createUser mutation will return back the User Type data we created as the resolved value. This is the mutation definition.

The next step is to define a resolver for the mutation similar to defining a resolver for a Query definition. We would create a new root property within the resolvers object called Mutation. Within the curly brackets is all of the mutation type definitions resolver methods.

Mutation resolvers also receive the same four arguments of parent, args, ctx and info. Inside of the resolver method we perform one of the mutations i.e. creating, updating or deleting data and then responding accordingly.

We would need to wire up the backend to perform the correct task.

The uuid library provides us a function to generate a random unique id's. To install this package we can run the following command in the terminal within the project directory:

```
:~$ npm install uuid
```

The some array method allows us to return a true or false value if the callback function matches any elements within the array. We can use the ES6 shorthand syntax for the arrow function to implicitly return a value.

We can use the throw new Error JavaScript syntax to throw an error message back to the client. If an error occurs this line would run and all code below within the resolver method will not be executed.

We can also use the push array method allows us to add the object to the end of an existing array of objects.

Finally, our resolver method must return back a value for the client and in the above case this would be the created user including the unique id assigned to the user object.

In the GraphQL Playground to perform a mutation we would write the syntax as seen in the below screenshot. We would start the query using the mutation keyword as we would do for query. Within the curly brackets we list out the mutation we would want to run. The syntax would look similar to the syntax for running a query.

users comments posts createUser × +

PRETTIFY HISTORY http://localhost:4000/ COPY CURL

```
1 mutation {
2   createUser(name: "John Doe", email: "j.doe@email.com"){
3     id
4     name
5     email
6     age
7   }
8 }
```

▶

```
{
  "data": {
    "createUser": {
      "id": "ebaf6f41-ad08-45c3-9d46-75e4ae51e682",
      "name": "John Doe",
      "email": "j.doe@email.com",
      "age": null
    }
  }
}
```

DOCS SCHEMA

users comments posts createUser × +

PRETTIFY HISTORY http://localhost:4000/ COPY CURL

```
1 mutation {
2   createUser(name: "John Doe", email: "j.doe@email.com"){
3     id
4     name
5     email
6     age
7   }
8 }
```

▶

```
{
  "data": null,
  "errors": [
    {
      "message": "Email taken.",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "createUser"
      ]
    }
  ]
}
```

DOCS SCHEMA

The data we get back from the mutation query matches up with the structure of our query i.e. we have a property for the mutation name we selected with all of the data we chose to select from the user to return back to the client.

If the email exists within the users array we would get back an error object with our message as seen in the second screenshot example above.

It is important to note that the data in our users array is not persistent and therefore if the server was to restart this would also reset the users data and the user we created would no longer exist in the array. This is why we have a real database to store persistent data.

To conclude, like with a Query there are two sides to any Mutation: the client operation and the server definition. Therefore, we need to define the mutation on the server in order to be used from the client. For the definition we define a new Mutation type and we list out all of the mutations we want to support. To setup a mutation we give it a name, setup the arguments and setup the return value. We would then setup a resolver method for the mutation ensuring that we return whatever the mutation expects to return. From the client perspective, the syntax to perform a mutation is similar to a Query whereby we use the mutation keyword followed by the mutation name we would want to perform. The client can select the return fields required from the successful mutation.

The Object Spread Operator with Node.JS

The object spread operator is a useful syntax which makes it easy to copy object properties from one object to another. To get this operator working with Node.js we would need to install a npm package and configure Babel to use it. You can learn more about the package by visiting the following link:

<https://www.npmjs.com/package/babel-plugin-transform-object-rest-spread> or <https://babeljs.io/docs/en/7.7.0/babel-plugin-transform-object-rest-spread>.

To install the package we would need to run the terminal command within the project directory:

```
:~$ npm install babel-plugin-transform-object-rest-spread
```

When we install a plugin we would need to actually tell babel to use the plugin by updating the .babelrc file. The plugins property will hold an array of all the plugins we wish to use, each plugin stored as a string leaving off the babel-plugin- from the plugin name.

.babelrc:

```
{ "presets": [ "env" ], "plugins": [ "transform-object-rest-spread" ] }
```

Babel is now all setup to use the plugin — spread operator syntax. It is important to note that a preset is nothing more than a collection of plugins all grouped together to provide some cohesive behaviour. We want an individual plugin which is why we have both presets and plugins listed.

After saving the changes to Babel we can restart the server by running the `npm run start` command in the terminal.

The goal of the spread operator is to be able to easily copy properties from one object to another which is something that we are doing manually thus far (in the examples) with our mutation methods. For example, we have the `args` object with various property on it and we copy it over to the `user/post/comment` object manually which is cumbersome. The object spread operator provides us with a much efficient way to get this all done. Below is an example on how the object spread operator syntax works:

```
const one = { location: 'London', country: 'UK' };  
const two = { population: 8900000, ...one };
```

We use the three periods `...` followed by the name of the object we want to spread out. This has the effect of copying all object properties from one object over to the other object. Therefore, taking the `user` object example we can shorten the syntax using the spread operator:

[src/index.js:](#)

```
... const resolvers = {  
  Mutation: { createUser(parent, args, ctx, info) { const user = { id: uuidv4( ), ...args }; }  
};
```

This allows for maintainable code which is easily scalable should `args` have more properties on it.

The Input Type

In GraphQL there is an advanced way to structure operation arguments especially where we have more than one operational arguments. This allows for more complex applications without having complex code.

Instead of listing out all of the arguments and the scalar values we would want to pass in one object which has all of the arguments and scalar values as properties. To get this done, we need to define an Input Type which has all of the arguments and scalar values and then reference the Input Type.

To define an Input Type within the Type Definitions we use the keyword `input` followed by a generic name we choose for these input. Within the curly brackets is the definition for all the properties that could exist on this input (argument) we are setting up.

[src/index.js:](#)

```
const typeDefs = `
  type Mutation { createUser{ data: CreateUserInput! } }: User!
  input CreateUserInput { name: String!, email: String!, age: Int }
`
```

We can create a single argument which we can name anything we would like (in the above this is named `data`) and set its type to the input type we created. We can only reference an input type and not a custom object type this is because our input types can only have scalar values.

We would also need to update our resolver methods to reference the arguments name when we make reference to the args parameter i.e. args.data. followed by the operation argument property name.

The queries in GraphQL Playground would now look like the below (making reference to data (or whatever we name the argument) and passing in the arguments within the curly brackets):



The screenshot shows the GraphQL Playground interface. At the top, there are tabs for queries (Q) and mutations (M). The 'createUser' mutation tab is active. Below the tabs, there are buttons for 'PRETTIFY', 'HISTORY', and a URL bar showing 'http://localhost:4000/'. To the right of the URL bar is a 'COPY CURL' button. The main area is split into two panels. The left panel contains the query:

```
1 mutation {  
2   createUser(  
3     data: {  
4       name: "John Doe",  
5       email: "j.doe@email.com"  
6     }  
7   } {  
8     id  
9     name  
10    email  
11    age  
12  }  
13 }
```

 The right panel shows the JSON response:

```
{  
  "data": {  
    "createUser": {  
      "id": "6b158027-7514-4b47-8e81-f696dbb7c3cf",  
      "name": "John Doe",  
      "email": "j.doe@email.com",  
      "age": null  
    }  
  }  
}
```

 A play button is located between the two panels. On the far right, there are vertical buttons for 'DOCS' and 'SCHEMA'.

To conclude, Input Types does not change the functionality of the application but instead it allows to have a simple definition for our Mutations and Queries and the ability to reuse the Input Types across other operations creating a better structure to the application code.

Deleting Data with Mutations

When deleting data it is important to be mindful of not just the actual thing we are trying to delete but also the associated data. For example, if we are deleting a user not only do we want to delete the user but we may also want to delete all posts and comments created by that user otherwise we may have invalid data causing errors when retrieving non-nullable fields.

The deletion of data is a straight forward process but we also must be considerate of the other data in our application. Below is an example code for the delete mutation.

[src/index.js:](#)

```
const typeDefs = `
  type Mutation { deleteUser{ id: ID! } }: User!
`

const resolvers = {
  Mutation: {
    deleteUsers(parent, args, ctx, info) {
      const userIndex = users.findIndex((user) => user.id === args.id);
      if (userIndex === -1) { throw new Error('User not found.')}
      const deletedUsers = users.splice(userIndex, 1);
      const posts = posts.filter((post) => {
        const match = post.author === args.id;
        if(match) {
```

```

        comments = comments.filter((comment) => comment.post !== post.id);
    };
    return !match;
});
comments = comments.filter((comment) => comment.author !== args.id);
return deletedUsers[0];
}
}
};

```

We use the `findIndex` array method is identical to the `find` array method although `find` returns the actual element in the array while `findIndex` returns the index number of the element in the array. If there are no match the method returns `-1` as the value because indexes are zero based.

In the above example, if we find a user we would delete that user and then delete all of that user's posts and comments. To remove data from an array we could use something like the `filter` array method or the `splice` array method.

The `splice` array method is called on the array and is passed two arguments. The first argument is the index where we want to start removing items from the array and the second is the number of items we would like to remove from the array. All of the removed items actually come back in an array as the return value which we can store in a variable.

Important Note: if we are hardcoding the array values, we should use the `let` variables (and not `const` variables) to be able to reassign the array values.

We can use the `filter` array method to keep the posts and comments that does not belong to the deleted user.

To conclude, it is really important to see that when we are working with data in GraphQL, we cannot just focus on the individual thing but we also have to focus on the graph of data i.e. the relationships.

A Pro GraphQL Project Structure

Thus far we have learned how to create a GraphQL API using Queries and Mutations; however, all of the code exists within a single file. This approach makes it difficult to find things in the code and therefore we would want a better project directory structure so that it is easier to continue to expand and stay organised. In this chapter we will explore a better structure by separating our code into their own files.

In the `src` directory we would create a new file called `schema.graphql` (or whatever we would like to call it). This file will contain GraphQL code. The `.graphql` extension will provide us with features such as GraphQL syntax highlighting if we installed the various plugins in Visual Studio code.

We can store all of our application Type definitions, example below:

src/schema.graphql:

```
type Query { ... }
```

```
type Mutation { ... }
```

```
input CreateUserInput { ... }
```

```
type User { ... }
```

We have the application schema as we had before but it now lives in its own separate file which makes it much more easier to find, manage and work with. In the index.html file we would remove the typeDefs variable entirely and reconfigure our GraphQL-yoga server.

src/index.html:

```
const server = new GraphQLServer ( { typeDefs: './src/schema.graphql', resolvers } )
```

The typeDefs will no longer reference the typeDefs variable, instead we would set it as a string and the string value is the path to our schema.graphql file where the path is relative to the root of our application. We now have a separate schema type definition file but our application would continue to work as it did before.

We would also notice that nodemon will not restart our server whenever we make a change to our

other files other than the index.js file because nodemon is not watching files with the graphql extension by default. Nodemon by default will watch changes made to files with the extension of .js, .mjs, .coffee, .litcoffee and json extensions. However, we can specify our own extensions using the -e or --ext flags. Therefore, we would update the package.json start script to:

package.json:

```
{ ..., "scripts": { "start": "nodemon src/index.js --ext js,graphql --exec babel-node" } }
```

This will now watch for all files with the .js and .graphql extensions for changes in order to automatically restart the server when it detects a change to these file types. We would need to restart the script again for the changes to take effect.

We would create a new file in the src directory called db.js which will be used to store our static database data. We would also continue to use this file even when we no longer work with static data and use start to use a database.

db.js:

```
const users [ { ... }, { ... }, { ... }, ... ];  
const posts [ { ... }, { ... }, { ... }, ... ];  
const comments [ { ... }, { ... }, { ... }, ... ];  
const db = { users, posts, comments};  
export { db as default };
```

This db object will contain all our data which we will export out so that our index.js file can import it. We can also convert the objects back from let to const variables since they are no longer going to get reassigned.

This now brings up another problem how do we take advantage of the separate db.js file. We can import the db variable into the index.js file and then access this variable; however, the problem is that the resolvers are going to live in their own files/folders. To solve this, we would need to analyse the third argument of ctx which gets passed to all of our resolver methods.

The context is something we can set for our API and the context is an object with a set of properties will get passed to every single resolver method. Therefore, we can setup the db object to be part of our context and that db object will get passed to every single resolver method regardless of where it is actually defined. We do this by adding a configuration called context to our GraphQL-yoga server.

src/index.html:

```
... import db from './db';  
const server = new GraphQLServer ( {  
  typeDefs: './src/schema.graphql', resolvers, context: { db }  
} )
```

Context value is an object which holds the things we want to setup on ctx i.e. we can choose whatever values we happen to need. Later on we would pass in our database connection, authentication, etc. Context is a very useful and important feature of GraphQL/GraphQL-yoga server.

On context we setup a db property and give it the value of the db import variable. We can use ES6 shorthand syntax for this and set the value to db since both the property name and value are named the same. In effect, we are now passing the db database object to all of our resolvers for our application regardless of where these resolvers actually live. This provides us with a tonne of flexibility allowing us to move the resolvers to different files and folders which is what we are going to end up doing. We would want to structure our application in a way that it does not constrain us as our application grows and context is a big part of achieving that.

Using the above example, all of our resolver methods now have an object stored on ctx and that object has a single property called db and the db object has three properties of users, posts and comments. Therefore, in our resolvers if we want to return some data we would use ctx.db. followed by the object/property we want e.g. ctx.db.user. We can take this one step further by destructuring the ctx argument to grab db directly which is a very popular thing to do, syntax below:

```
Query: { users(parent, args, { db }, info) { ... return db.users } }
```

We would need to refactor all our existing code to now use the context db property wherever we access the database objects i.e. users, post and comments in the resolver method code. We now have a setup where our resolver methods do not rely on the global db variable as it now gets passed in via the ctx which makes it easy to break these resolver methods out into their own files and folders.

To separate our resolvers methods, we would create a new subfolder directory within the src directory called resolvers. Within this directory we should have files for each root property i.e. Query, Mutation, etc. For example:

src/resolver/Query.js:

```
const Query = { user(parent, args, { db }, info){...}, ... };  
export { Query as default };
```

src/resolver/Mutation.js:

```
const Mutation = { createUser(parent, args, { db }, info){...}, ... };  
export { Mutation as default };
```

src/resolver/User.js:

```
const User = { posts(parent, args, { db }, info){...}, ... };  
export { User as default };
```

We would shift the code away from index.js into their own files. We would move only the object into the files and not the root property name i.e. everything within the curly brackets and store it in a variable. We would then export the variable as the default export.

Now that we have all our resolvers in separate files, we need to find a way to bring them into our index.js resolvers object. To do this we would need to import these files and then we can set them up in our resolvers const variable object or alternatively we could set them up in our GraphQL-yoga server resolvers property by setting the value as an object passing in the imported variables.

The final index.js file would now look like the below examples depending on the method chosen for setting up the server resolvers property:

src/index.html:

```
... import db from './db'; import Query from './resolvers/Query';  
import Mutation from './resolvers/Mutation', ...  
const resolvers = { Query, Mutation, User, Post, Comment };  
const server = new GraphQLServer ( {  
  typeDefs: './src/schema.graphql', resolvers, context: { db }  
} )
```

alternatively,

src/index.html:

```
... import db from './db'; import Query from './resolvers/Query';  
import Mutation from './resolvers/Mutation', ...  
const server = new GraphQLServer ( {  
  typeDefs: './src/schema.graphql',  
  resolvers: { Query, Mutation, User, Post, Comment },  
  context: { db }  
} )
```

Our application is now all re-factored using a much better project directory structure which is the preferred approach for real world production grade GraphQL APIs. The purpose of index.js is now to load in the different files and bootstrap our application.

To conclude, we now have a setup where our Type Definitions are now in their own file called `schema.graphql`, we have context for the application which holds values for the application which are universal i.e. things that should be shared across our application e.g. database connection and finally our resolvers methods in their own folder and files that have access to the context values via the `ctx` arguments which gets passed to all resolver methods.

Updating Data with Mutations

Similar to the Create and Delete mutations the process for the Update mutation is exactly the same. We would define the mutation in the Type Definition (schema) and then the resolver method for the mutation. Below is an example code for an Update mutation using the new project structure:

`schema.graphql`:

```
type Mutation: { ..., updateUser(id: ID!, data: UpdateUserInput! ): User! }  
input UpdateUserInput { name: String, email: String, age: Int! }
```

`src/resolvers/Mutation.js`:

```
const Mutation = { ...,  
  updateUser(parent, args, { db }, info) {  
    const { id, data } = args;  
    const user = db.users.find((user) => user.id === id);  
    if(!user) { throw new Error('User not found'); };  
    if (typeof data.email === 'string') {
```

```

    const emailTaken = db.users.some((user) => user.email === data.email);
    if(emailTaken) { throw new Error('Email in use. '); };
    user.email = data.email;
  };
  if (typeof data.name === 'string') { user.name = data.name };
  if (typeof data.age !== 'undefined') { user.age = data.age };
  return user;
}
};
export { Mutation as default };

```

The first action is to locate the user to update by its id within the dataset else we would throw an error to state that the user does not exist.

There are three alternative methods in which we can use the resolver args argument. The first method is not to reference args in the resolver arguments but call it in the method:

```

updateUser(parent, , { db }, info) {
  const user = db.users.find((user) => user.id === args.id);
}

```

The second method is to destructure args by grabbing its object properties:

```
updateUser(parent, { id, data }, { db }, info) {  
  const user = db.users.find((user) => user.id === id);  
}
```

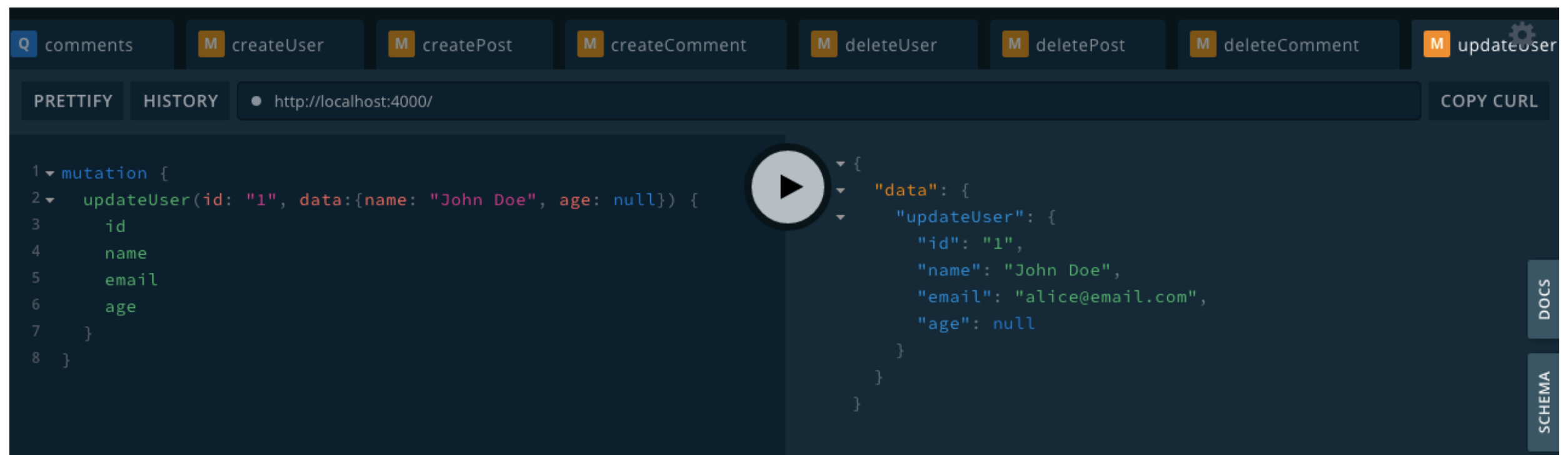
The final method is the one used in the update mutation example above where we would leave the args argument in place and then destructure it within the resolver method body:

```
updateUser(parent, args, { db }, info) {  
  const { id, data } = args;  
  const user = db.users.find((user) => user.id === id);  
}
```

This is purely a stylistic choice and there is no right or wrong method to use the args resolver argument.

Once the user has been found to update, we would look at the data to see which properties to actually update and update them. Using the typeof JavaScript function, we can check whether the data is the correct data type before updating. For email we would verify that there are no other user that has the email to update to. For the age we would check that the age is not undefined because Int and Null values are both valid values. The GraphQL will prevent the age from being anything other than a Int or Null value. If a valid value is provided for age then age would be updated.

To perform an update mutation in GraphQL Playground, the query would now look like the below:



As we can see from the above we have updated the name and age of the user with id of “1” using the Update mutation. We do not necessarily have to provide any arguments to the Update mutation because the three operation arguments are optional. If we provide no arguments this would successfully update the user without any changes returning the user data back. If we were to update the email to an existing email we would receive the error message of “Email taken”.

This concludes the topic on mutations and should now be able to support the basic CRUD (Create, Read, Update and Delete) operations in GraphQL which is an important first step of creating our very own GraphQL APIs.

Section 3: GraphQL Basics: Subscriptions

What is a Subscription?

The third operation we can perform on GraphQL is called subscription. Subscription allows us to subscribe to data changes. This will allow us to be notified of changes to data automatically without having to manually fire off a Query operation to get the latest data.

GraphQL subscriptions use Web Sockets behind the scenes which keeps an open channel of communication between the client and the server. This means that the server can send the latest changes to the client in real time. This feature is very useful for real time chat application, ordering applications, notifications and more.

This is the last of the three main operations provided by GraphQL.

GraphQL Subscription Basics

In essence the Subscription operation is similar to the Query operation because its main concern is allowing the client (whether it is GraphQL Playground or a web application) to fetch the data that it needs. The difference between the two is how the data is fetched.

When we make a Query operation, we send the query off to the server and the server responds a single time with all of the data at that point in time. If one of the data got created, deleted or edited the client would not be notified. It will be up to the client to make the same request later to check for changes in the data. This ends up leading to server polling where we run the operation from our client

every minute in an attempt to keep our client up to date (i.e. attempt to show real time data to the user). This is not ideal because it is expensive and requires us to perform operations every minute or so in which case a lot of the times the data has not changed and is wasting resources.

With a subscription we actually use Web Sockets to keep an open connection between the client and the server. This allows the server to transmit data directly to the client. Therefore, if data was to change the server can perform the change and push the change in real time down to all of the subscribed clients so that they can get the new data and render it to the User Interface (UI). This in effect keeps their application up to date creating a real time application where the user is viewing the latest data as it changes.

The process for setting up a subscription is a little more complex than setting up a query or a mutation. To setup a subscription we would first need define the subscription within the schema.graphql file Type Definition.

schema.graphql:

```
type Subscription { count: Int! }
```

We would use the type Subscription keyword and within the curly brackets are the various subscriptions we would like to support. The syntax is very similar i.e. we provide a name for the subscription, optional operation arguments and finally what data comes back from the subscription.

We would then need to setup a new root property on the resolver. Therefore, we would create a new file in the resolvers directory called Subscription.js which will hold all of the subscription resolver methods which we can export as the default and import into our index.js file.

src/resolvers/Subscription.js:

```
const Subscription = { };  
export { Subscription as default };
```

src/index.js:

```
import { GraphQLServer, PubSub } from 'graphql-yoga';  
import db from './db';  
import Subscription from './resolvers/Subscription';  
const pubsub = new PubSub();  
const server = new GraphQLServer({ ..., resolvers: { ..., Subscription }, context: { db, pubsub } });
```

Before we add to the subscription resolver object in the Subscription.js file, we would need to setup one thing in our GraphQLServer. If we bring up the GraphQL-Yoga documentation (<https://github.com/prisma-labs/graphql-yoga>) we will notice with one of the libraries that GraphQL-Yoga uses behind the scenes called graphql-subscriptions. This is the library we would have to use to get things to work with our GraphQL subscription operation. This library provides us with a single PubSub utility (pub for published and sub for subscribe) features to allow us to communicate around our application.

We will therefore be grabbing a named export called PubSub from graphql-yoga. This is the constructor function allowing us to create a new instance of the PubSub utility which we can store in a const variable. This is something we would want to pass to all of our resolvers. We would need to use the pubsub directly from our subscription methods which means that we would need to add pubsub to GraphQLServer context property. This completes the setup of our index.js file and we can now head over to the Subscription.js file to setup a new subscription.

For every subscription we would have to setup a new property which would need to match up with the subscription name in our Type Definition.

src/resolvers/Subscription.js:

```
const Subscription = {  
  count: {  
    subscribe(parent, args, { pubsub }, info) {  
      let count = 0;  
      setInterval(() => {  
        count = count++; pubsub.publish('count', { count: count });  
      }, 1000);  
      return pubsub.asyncIterator('count');  
    }  
  }  
};
```

Unlike Queries and Mutations the value for a subscription is actually not a method but rather an object. On this object we setup a single method called `subscribe`. The `subscribe` method is what runs every time someone tries to subscribe to subscription property i.e. `count` in the above example. The `subscribe` is a regular resolver method and gets called with all of the same arguments of parent, args, ctx and info. The only thing we would be using off of ctx is the `pubsub` utility we setup in the `index.js` file which we can destructure.

When it comes to the return value for a subscription things a little different than with Queries or Mutation resolvers. The return value for a Query or Mutation is a match of what was defined in the Type Definition (Schema). With a Subscription we do not return what was defined in the Type Definition (Schema). Instead, we return something that comes from the `pubsub` utility.

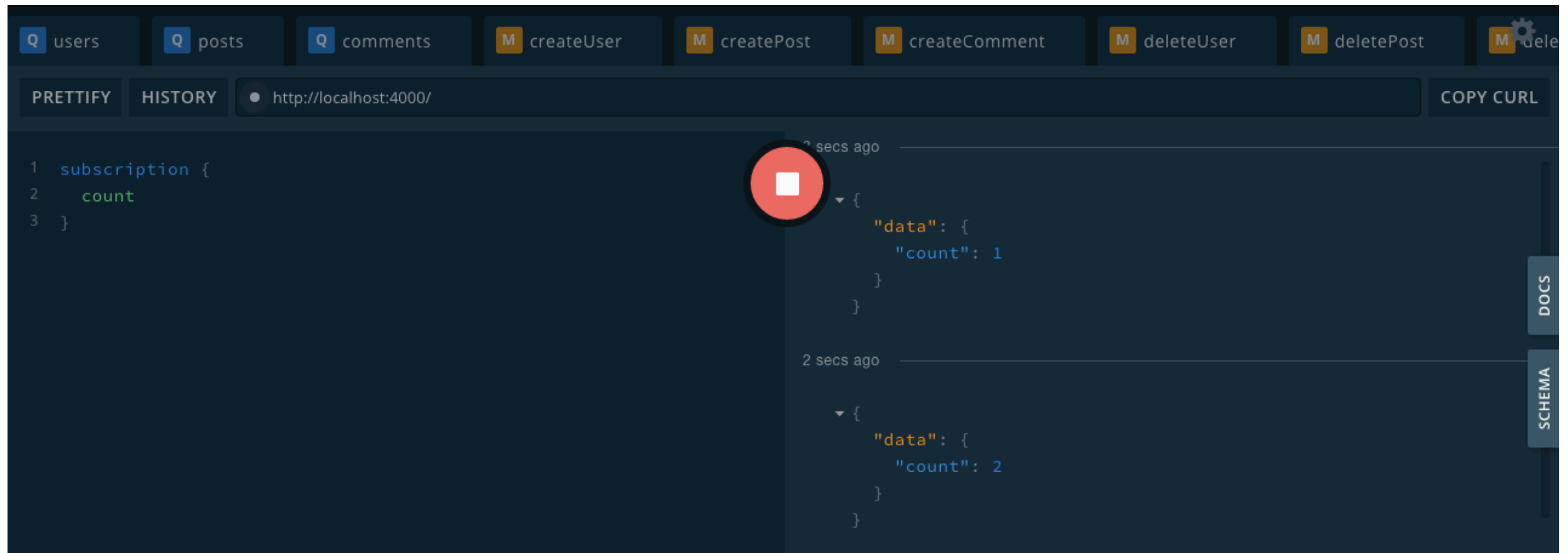
We call a method on `pubsub` called `asyncIterator` which is a function which takes in a single argument. The argument is a string and the value here is what's called a channel name. We can think of a channel name of a chatroom name. From here we have a valid subscription but it is not going to publish any changes.

The `setInterval` function allows us to run some code after a number of milliseconds where 1000 milliseconds = 1 second. This function takes in two arguments the first is the callback function to run and the second is the number of milliseconds.

In the callback function, `count++` will increment the value by 1.

The `pubsub asyncIterator` method sets up the channel while the `publish` method allows us to publish new data to all of the subscribers. The `publish` method takes in two arguments, the first is the channel name and the second is an object where we specify the data that should get sent to the client. The second object argument is where the data must match up with what was defined in the Type Definition for the subscription.

In the GraphQL Playground to subscribe to a subscription we would use the following syntax as seen in the screenshot below:



We would use the `subscription` keyword and in the curly brackets we would select the name of the subscription we would like to subscribe to. If the subscription took arguments we would provide them.

If the subscription returns an object we would provide a selection set. In this example the count subscription returns an Integer and so the subscription name is all we need. When we run the query we would see GraphQL Playground is listening for changes and waiting for the server to push data down to the client and when it does that new data will show up.

We would notice in the example above, every second a new data comes in incrementing the count. This data is a data object with a count property and the value of that property is the latest count stored as an integer. This will continue to run for as long as we would let it.

To conclude, we should now have an introductory example allowing us to get comfortable with the syntax and have the basic knowledge for setting up a Subscription operation in GraphQL which will allow us to setup advance subscription to the latest data whenever a data changes via a mutation.

Enumerations

The enums type is another tool within our tool belt allowing us to better model our application data. Enum is short for enumeration and is a special type that defines a set of constants. This type can then be used as the type for a field similar to scalar and custom object types. Values for the field must be one of the constants for the type.

For example, We may have a UserRole enum which has a list of potential values such as standard, editor and admin. When we go to find the user, we can use the

enum as the type for one of the fields e.g. role: UserRole! This will enforce that the value for role is one of the three values of UserRole and cannot be anything else. Enums allow us to represent as many states as we need and enforce one of those states.

When the data is coming from the client it is even more important to use an enum because we can actually enforce that it is of one of the constant types. Below is an example code:

schema.graphql:

```
enum MutationType { CREATED UPDATED DELETED }  
type PostSubscriptionPayload { mutation: MutationType! data: Post! }
```

We would use the enum keyword followed by the enum type name. Within the curly brackets we would define the potential enum constant values. It is a typical convention to use an Uppercase character set for all of the values although it is not enforced. We can then use the enum type and now the return value must be one of the three constant values.

Enums allows us to easily catch typos and inconsistencies throughout our application.

To conclude, enums allows us to represent a set of constants and then any fields of that type must have a value equal to one of the constants. This is great when we are modelling data and we have a set of standard values that we know about ahead of time. If working with two values we could get away with using boolean values while three or more we would use enums.

Section 4: Database Storage with Prisma

What is Prisma?

Referring back to the diagram from Section 1 we can see the representation of GraphQL with our application. However, the question we now need to focus on is how do we connect our backend NodeJS with the database of choice?



We would need a tool which will facilitate that communication. Therefore, when someone sends a mutation off to the server, the server needs to actually write to the database. When someone sends a query asking for data the server needs to actually read from the database as opposed to reading from a static file. How would we get that done?

There are a few options one of them being Prisma.

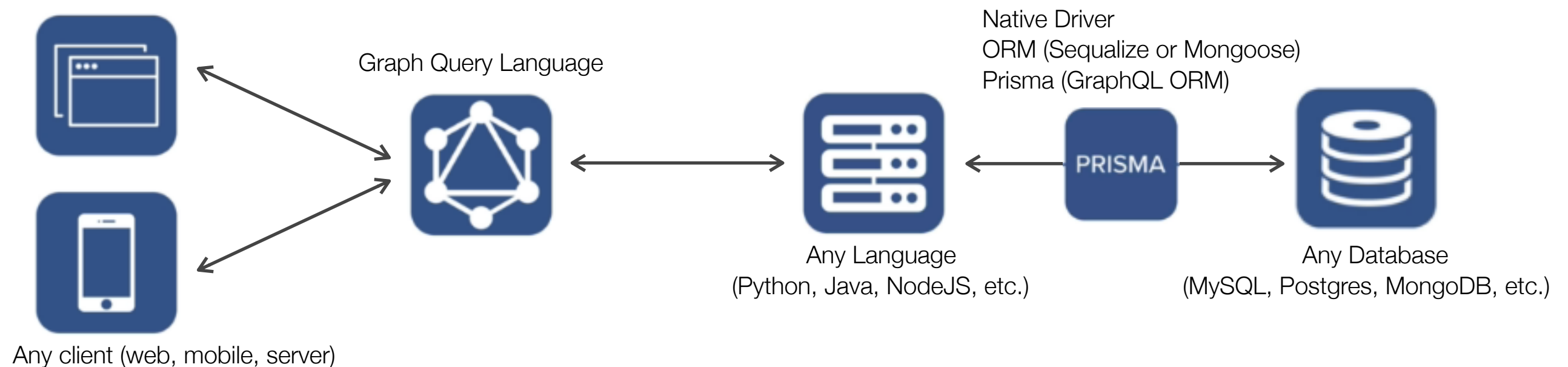
The first option is to use a native driver and all popular databases have native drivers for NodeJS. These are essentially npm libraries that make it easy for us to connect our backend (NodeJS) to those databases. Native drivers are very barebones implementations which allow us to perform all of the queries necessary to read and write data but we do not get nice to have features such as migrations, data validations, map models and set relationships between the data. If we go with Native drivers we would end up doing way more work than what is really necessary.

The second options is to use an ORM (Object Relational Mapping) for example Sequelize or Mongoose. Sequelize is a great NodeJS ORM for connecting Node to an SQL database while Mongoose is a great NodeJS ORM for connecting Node to MongoDB NoSQL database. ORM allows us to have those nice to have features.

Prisma is an ORM for Node. One of the great features of Prisma is that it is database diagnostic. If we were using the Sequelize library we would have to use an SQL database and if we use the Mongoose library we would have to use the MongoDB database. If we decide to switch from MongoDB to MySQL or vice versa we would have to re-write most of our application because the libraries are so different. With Prisma we would not need to do that as it supports every major database out there.

Currently Prisma has support for MySQL, Postgres and MongoDB and they are working to support other databases such as Elasticsearch and Cassandra. This would mean we could choose the database we would want or switch between databases without needing to change much code (i.e. only the database connection setup).

All of this works because Prisma wraps our database up and exposes it as a GraphQL API. This allows us to read and write to the database regardless of what database we are using i.e. SQL or NoSQL. This means that our NodeJS backend can read and write from the database using GraphQL.



We would notice that anytime where we have communication between the different layers we have GraphQL. So if the client wants to communicate with the NodeJS server it uses GraphQL and if the NodeJS server wants to communicate with the database it uses GraphQL as well using the Prisma ORM.

Because we are using GraphQL between the client and the server and between the server and the database the server itself actually becomes a whole lot less important. This allows us to reduce the amount of code and complexity of that code. The server acts as a thin layer between the client and the database and so it almost feels as though the client has direct access to the database. The server is still very important for things such as authentication and data authorisation.

To conclude, Prisma is just an ORM which works with all databases and it makes it really easy to expose access to the database to a client in a secure and efficient way.