



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.

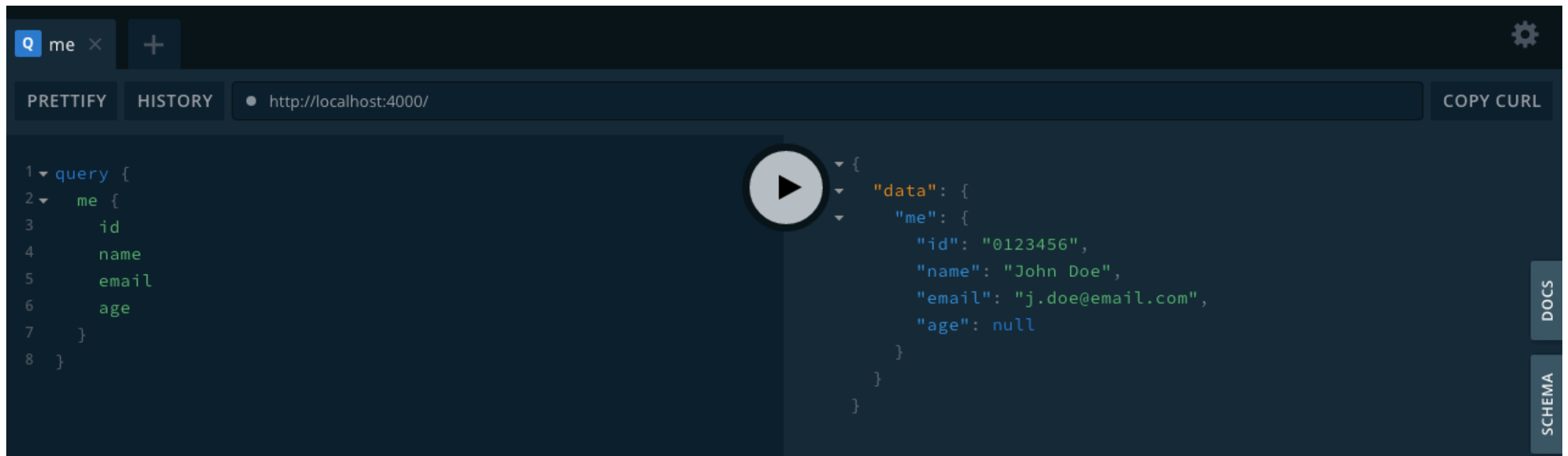
Within the curly brackets we define all of the fields that make up the custom Type. This will look similar to defining Query definitions.

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/'. On the right, there's a 'COPY CURL' button. The main area is split into two panels. The left panel contains a query:

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

. The right panel shows the JSON response:

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

. A large play button is in the center of the split view. On the far right, there are vertical buttons for '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!';
} } };
```

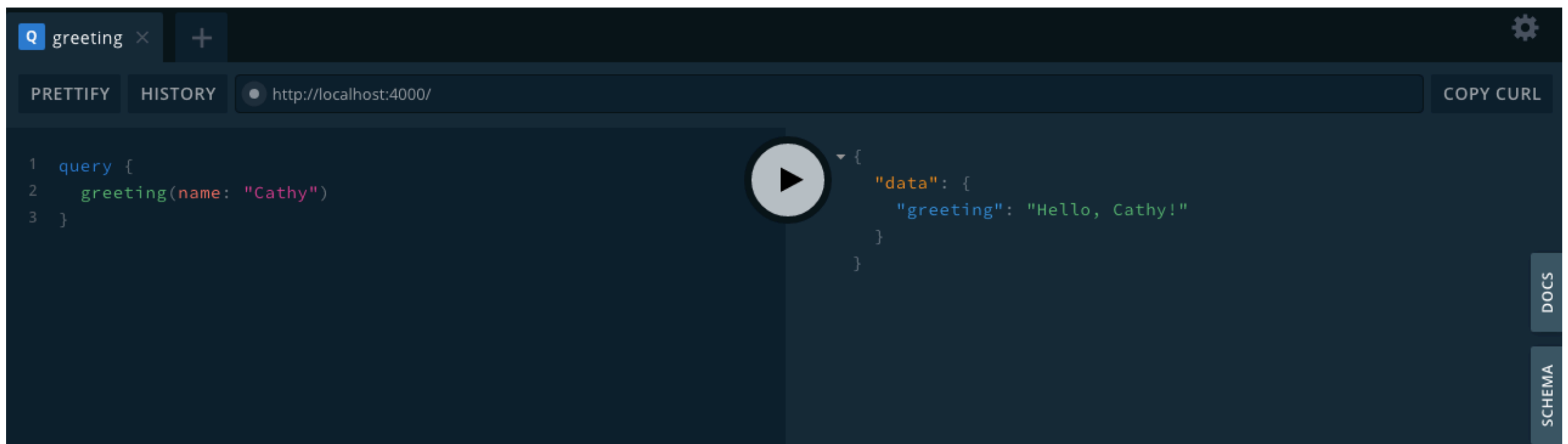
The query will return a scalar type at the end of the day; however, we can determine what arguments this query should accept by adding round brackets after the query name and before the colon. This is where we define/list all of the arguments that could be passed along with the query and define whether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

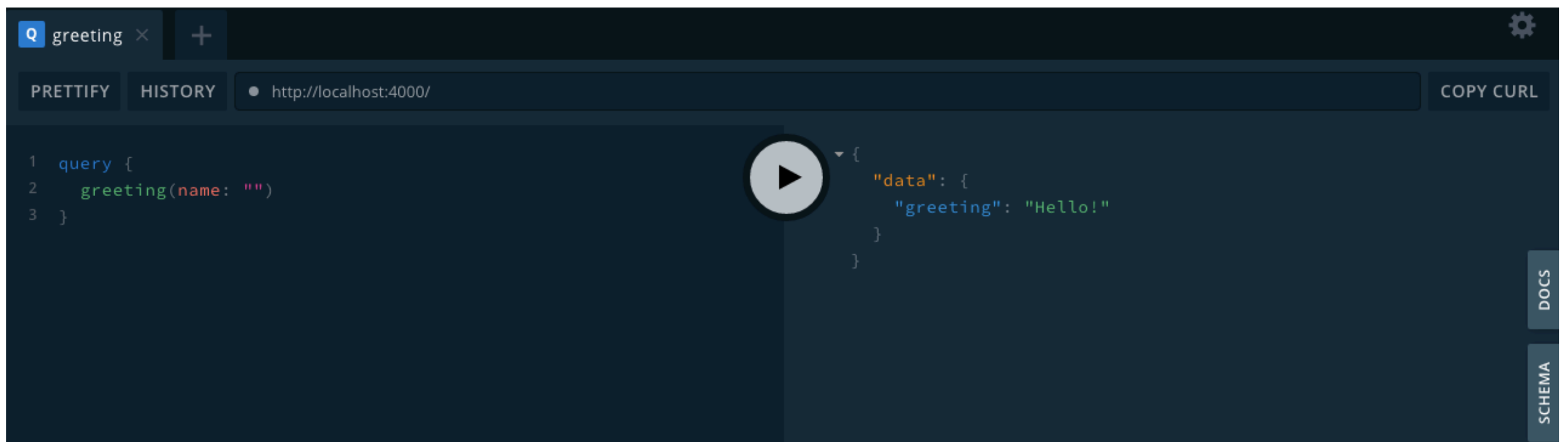
We would then setup the resolver method for this query. It is up to the resolver method to use the data to make some sort of dynamic response. There are four arguments in the specific order that get passed to all resolver functions and they are as follows:

- ◆ 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:



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.

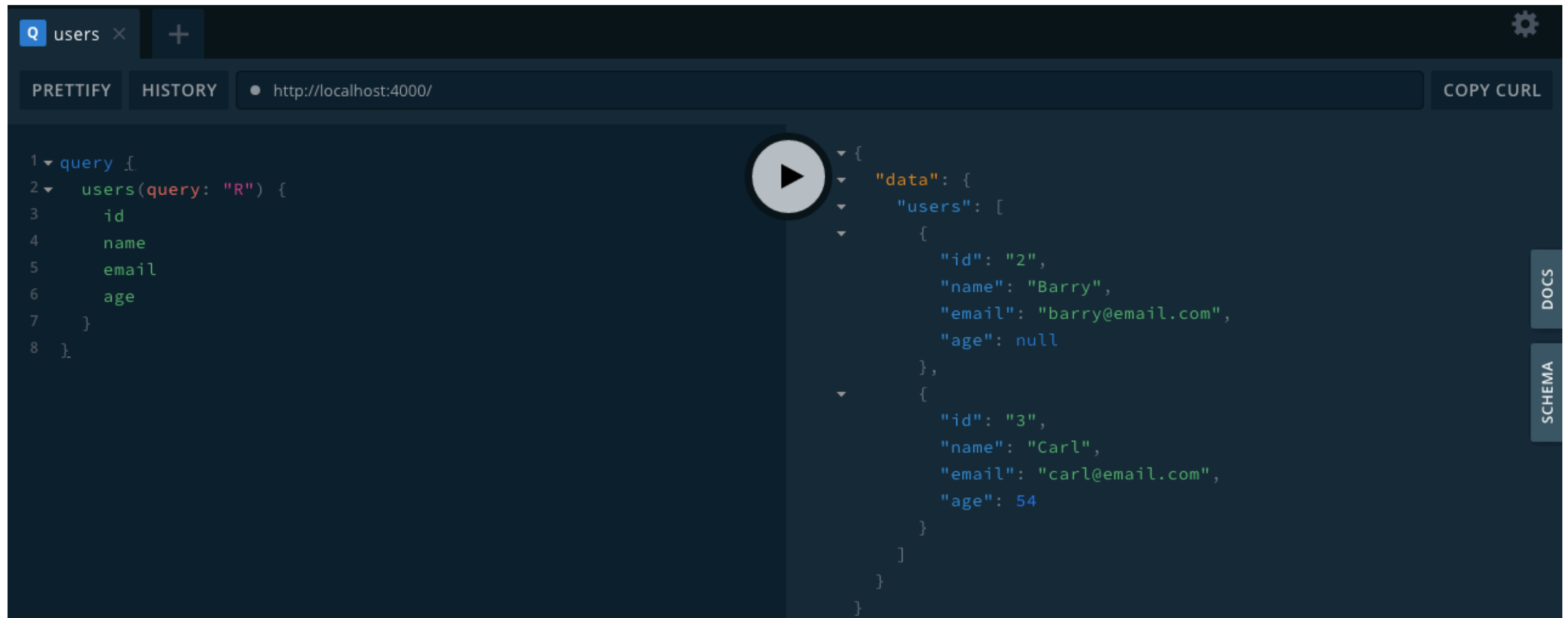
We can achieve this by using operation arguments and array data types. Below is an example code:

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 'PRETTIFY' button, a 'HISTORY' tab, a URL bar showing 'http://localhost:4000/', a 'COPY CURL' button, and a 'DOCS' button on the right side.

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 active tab is 'createUser'. The URL bar shows 'http://localhost:4000/'. The query editor on the left contains the following GraphQL mutation:

```
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 response editor on the right shows the JSON response:

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

On the right side of the interface, there are 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.

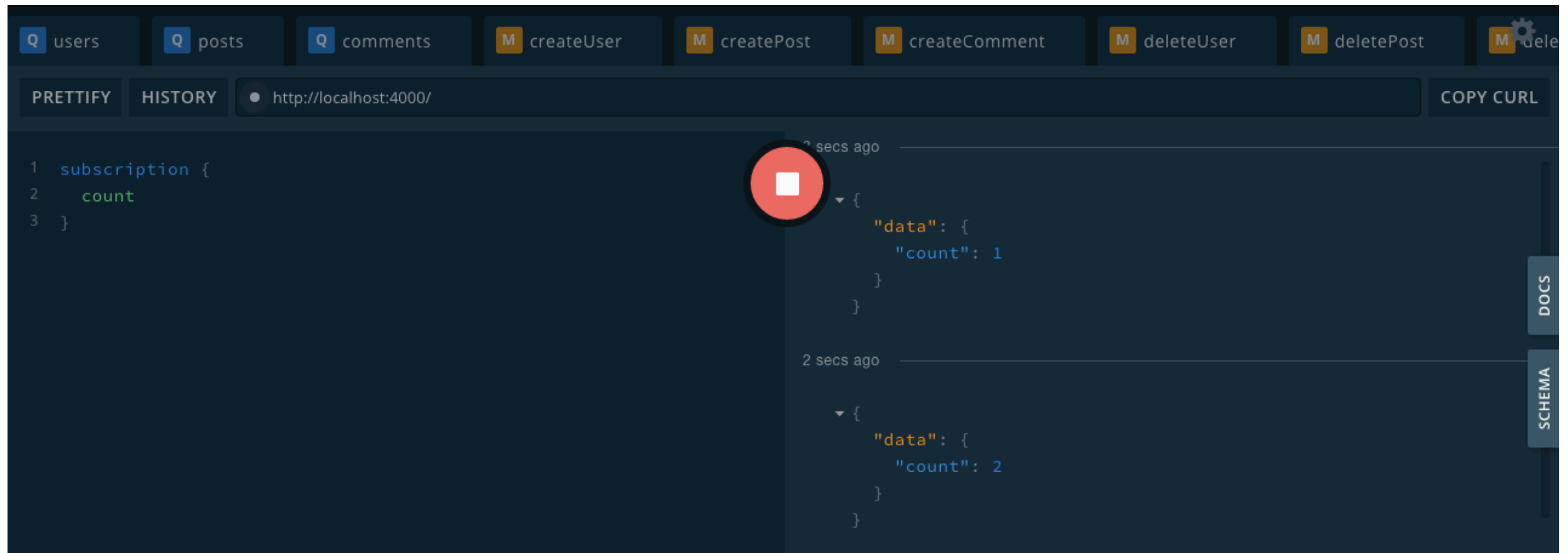
We call a method on `pubsub` called `asyncIterator` which is a function which takes in a single argument. The argument is a string and the value here is what's called a channel name. We can think of a channel name of a chatroom name. From here we have a valid subscription but it is not going to publish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

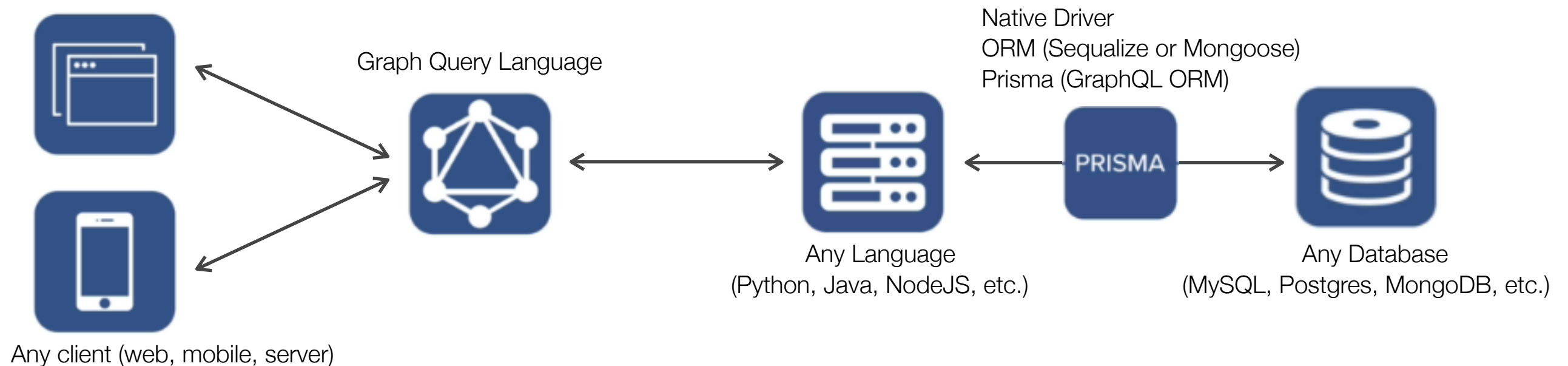
The first option is to use a native driver and all popular databases have native drivers for NodeJS. These are essentially npm libraries that make it easy for us to connect our backend (NodeJS) to those databases. Native drivers are very barebones implementations which allow us to perform all of the 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.

Prisma Setup

The first step requires to setup a database of choice. PostgreSQL is a SQL database that pairs really well with GraphQL. All SQL databases in general pair really well with GraphQL because with GraphQL we need to be explicit about the exact fields and their type definitions and so it is easy to match with the SQL database.

To install Postgres we can visit <https://www.heroku.com/> to get a free tier of a basic database. We could also use any other alternative methods of installing PostgreSQL on our machines such as visiting <https://www.postgresql.org/>.

If we are using Heroku, we would need to create a new app. There are plenty of features for Heroku but we are only going to use a very small subset of them. To get started head over to the Overview tab for the application to install add-ons. We would want to install the free tier Heroku Postgres add-on. Once the database has been created we would need the credentials in order to connect to the database which we can find in the settings tab.

The second step is to install pgadmin from <https://www.pgadmin.org/> which is a GUI tool allowing us to interact with the Postgres database without having to use any code. The GUI will allow us to easily see what data we are working with and how data is changing over time. We can setup the connection to our Heroku Postgres database with pgadmin by using the Add New Server option within pgadmin.

Below is the table to match the Heroku Database Credentials to the pgadmin Add New Server setup Connections fields.

PgAdmin Connections Fields	Heroku Database Credentials
Host name/addresses	Host
Port	Port
Maintenance database	Database
Username	User
Password	Password

We can tick the Save Password? checkbox to save the password without having to type in the password every-time we want to connect to the database from pgadmin. We can save the New Server setup. We should now see a Databases for the connection in the Browser sidebar and if we expand this we should see all of the databases on the cluster. We can only access our own databases from the cluster. To find our database in pgadmin we would copy the database name (this can be found on the Heroku Database Credentials) and use the find on page in the browser to search for the database by name in order to select it.

We should now have a database setup and a way to connect to that database from a GUI. The final step is to install a tool called Docker from <https://www.docker.com/get-started>. This is a dependency which we must have installed on our machine in order to use Prisma. There are two sides to Docker, the person creating the application container and the person who is using the container to run the application. We are on the latter side and therefore do not need to learn the ins and outs of Docker. You require a Docker account in order to download the desktop application. On MacOS the application runs in the top bar with Docker is running status while on Windows we need to have the Docker Quickstart Terminal running. We now have everything necessary installed to use Prisma.

Prisma 101

We should have a PostgreSQL database setup, pgadmin running with a connection to the database and finally docker started and running in the background in order to continue with using Prisma in our projects. We can learn more about Prisma on its website <https://www.prisma.io/> which provides both guides and documentations.

Prisma is installed as an npm package by running the terminal command. We use the -g flag to install the package globally:

```
:~$ npm install -g prisma
```

To check whether Prisma was installed correctly we can run the following terminal command. This should return the version of Prisma installed on our machines:

```
:~$ prisma -v
```

After successfully Prisma we can now use Prisma commands in our projects. We can create a new project directory and run the following command:

```
:~$ prisma init prismaProjectName
```

The third argument is the name of the project which we can choose whatever name we would like. This would create a sub-folder within our project directory using that third argument name. Prisma would then ask a few questions about our project. We would want to connect to an existing database selecting the type of database i.e. PostgreSQL and finally does our database contain any existing data which we would select no. From here we can provide our database connection from Heroku.

The final question will ask whether to use SSL. External connections for Heroku require SSL and so we must answer Y (yes) to this final question. Depending on the Prisma version, the latest versions will ask to select the programming language for the generated Prisma client — for this question we would want to select Don't generate. We are going to build out our own Prisma client to understand how Prisma works.

This would now create three files in our `prismaProjectName` directory called `datamodel.prisma`, `docker-compose.yml` and `prisma.yml`. In the terminal we also see the next steps required to set things up.

The `datamodel.prisma` (previously `datamodel.graphql`) file is nothing more than a set of type definitions for GraphQL similar to what we have in the `schema.graphql` file. Prisma uses this file to determine our database structure and will create a new table for all of our custom object types; therefore, the GraphQL Type Definitions end up changing the structure of what our database looks like.

The `prisma.yml` file has a `.yml` extension. A `.yml` file is similar to a JSON file i.e. it holds a set of key:value pairs and is a great language for when it comes to configuration. The indentation for `.yml` files are really important for properties and sub properties.

The `docker-compose.yml` file starts up the docker container and is the complex file out of the three. This file contains the version for the docker-composed file syntax we want to use. In the services

property we have a single service of prisma. The prisma object holds all the configuration for this service. The prisma image property is the docker container that someone else has created which we are going to use. The restart property set to always will make sure to restart the process when we deploy new changes to it. The ports property is the port we would use for the docker container. The environment property holds all the prisma configuration properties for how Prisma should work.

We have to make a few changes to this file before we deploy it. First we would remove the schema property because it is not necessary. We would then need to make sure that the `ssl: true` property is within the configuration settings.

To deploy our application we would need to follow the 4 steps defined in the terminal:

1. Open folder: `cd prismaProjectName`
2. Start your Prisma server: `docker-compose up -d`
3. Deploy your Prisma service: `prisma deploy`
4. Read more about Prisma server:
<http://bit.ly/prisma-server-overview>

The `docker-compose up -d` command will download and install all of the code that is necessary to actually run Prisma and then start up the Prisma application in our virtual machine. This process does not deploy the code in any way whatsoever. We have to run `prisma deploy` in order to actually deploy our changes to the process. Therefore, as we make changes to our schema we are going to need to deploy for Prisma to actually reflect those changes in the database.

Once it has deployed we can visit the localhost URL to see the instance of GraphQL Playground. This time though the GraphQL Playground is connected to the GraphQL API provided by Prisma itself. This means that if we were to use a Mutation it would actually write data to our Postgres database and if we were to use a Query it would be reading data from the database and so on. Prisma does all of the heavy lifting for us. All we have to do is define in `datamodel.graphql` the various types and fields we would want and Prisma goes through the process of automatically creating the Queries, Mutations and Subscriptions necessary to get everything working with the database chosen.

Exploring the Prisma GraphQL API

The Prisma GraphQL API auto-generates Queries, Mutations and Subscriptions for us based on the `datamodel.prisma` file and this allows us to interact with our database. As we perform Mutations we are actually writing to the database and when we perform Queries and Subscriptions we are actually reading from the database.

If we take the example of the Type User we would see in the GraphQL Playground DOCS tab the auto-generated Queries for fetching single and multiple users, Mutations for creating, updating and deleting a user (including other ways of mutations for the user type) and finally a Subscription to subscribe to user changes whether created, updated or deleted.

If we click on one of the auto-generated Queries, Mutations or Subscriptions this will open the Type Details where we are also able to see how it works (as seen in the image below).

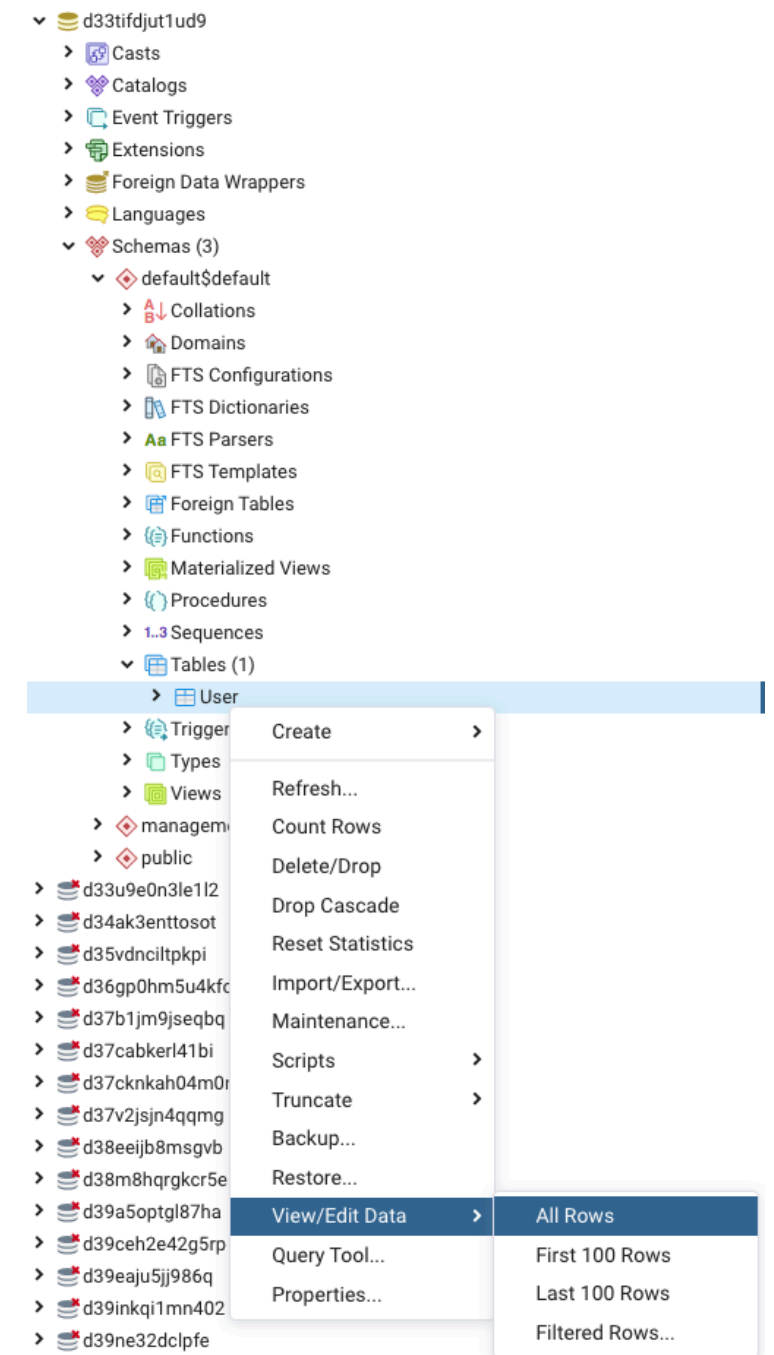
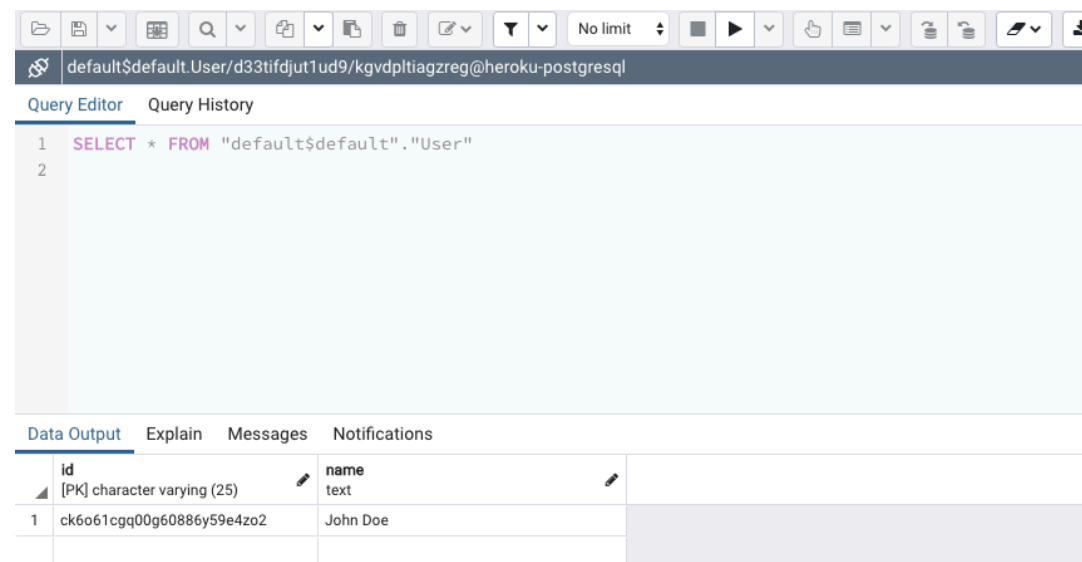


We would need to refresh the Schemas to see the latest changes and we can do this by right clicking on Schemas and selecting the Refresh option within our connected database. We would only do this when we actually make a change to the data structure.

If we expand the Schemas we should be able to see the various schemas we have and by default our data is stored under the default\$default schema.

If we expand the Tables within the default\$default Schema we should see all of the tables for every single model (Type) we defined. To view the data in a nice way as seen below we can right click the table name hover over View/Edit Data and selecting the All Rows option. We can use the Play button to re-run the query again if the data changed.

We can clearly see with Prisma that we are able to easily manipulate the data in our database using a GraphQL API.



There is nothing magical about how Prisma sends data back. It will look exactly the same as it looked when we were using GraphQL Playground with our own application (see previous sections). The benefit of using Prisma is that we do not need to create the Queries, Mutations or the Subscriptions because Prisma does that automatically for us.



Getting familiar with how to read the Type Details definitions within the GraphQL Playground DOCS tab is important and should be something we should get use to looking at to understand how the different operations work. This is an important skill to learn and the more we familiarise ourselves with how to read the schema the easier it will become to work the the Prisma API.

We now have a way to perform all of the CRUD operations using the auto-generated GraphQL API provided by Prisma. We now have a little taste of what the Prisma API actually provides us. It is also important to note that the Query, Mutations and Subscriptions will be same even if we used a different database (i.e. we do not need to change the code). Prisma abstracts away the implementation details of our database allowing us to focus on what our application needs to do.

Adding Type to Prisma Datamodel

The `datamodel.prisma` file is used by Prisma to determine two very important things. First, this file is used to determine the database structure — the only reason we have a `User` table with `id` and `name` as fields is because the `User` Type was setup this way. Secondly, this file is used for generating the GraphQL API Schema i.e. Queries, Mutation and Subscriptions.

It is important to note the `User` Type is automatically generated as a default type when we first setup a Prisma project using the Prisma CLI command. We can edit the `datamodel.prisma` file to edit existing Types and add our own custom types. We can continue to use the same GraphQL syntax as we have used previously although `.prisma` files allow for an alternative syntax. It is important to note that we should not change the `.prisma` extension to `.graphql` otherwise this will cause errors in the terminal when we push the code up to our docker container. Below is an example update to the `User` Type:

`prisma/datamodel.prisma:`

```
type User {  
  id: ID! @id @unique  
  name: String!  
  email: String! @unique  
  createdAt: DateTime! @createdAt  
  updatedAt: DateTime! @updatedAt  
}
```

When we make changes to the `datamodel.prisma` file we need to push this up to our Prisma server by running the following command in the terminal within our Prisma project directory:

```
:~.../prismaProjectName$ prisma deploy
```

When we run the command we would notice an error in the terminal. This is because we re creating a required field but there are already nodes (data) present that would violate this constraint. To work around this we can go into PgAdmin and delete all records within the table and then pushing the new `datamodel.schema` schema up to our Prisma server to apply the new changes to the tables.

In PgAdmin we should be able to see the changes to the database with the new column fields added to the User table. This would also change the GraphQL API which we can see in the GraphQL Playground's DOCS tab.

A GraphQL directive is a way to modify the behaviour of a field. Directives are usually something we would need to create and then use; however, Prisma provides us with some built in directives such as `@unique`, custom types and many more to make our life a little easier.

If we want to use a directive we use the `@` sign followed by the directive name for example `unique`, `id`, `createdAt` and `updatedAt`. Some directives can also take in arguments.

We would use the same normal syntax to create fields and relationships. When we deploy the changes if we have a relationship Prisma will auto-generate a Relationship table and not Type table for

example a if we created a relationship between a User and a Post this will create a User* (Type), Post (Type) and a PostToUser (Relation) tables. This will allow Prisma to setup the necessary tables to create a relationship between the User and Post.

Important Note: Prisma has changed the way it creates the tables in the latest version release. We would no longer see the `_PostToUser` table; however, the relationship will exist between the Post and User via an `Author` field in the Post table containing the ID/Email (unique fields of the User type) of the author as the foreign key.

OUTPUT **TERMINAL** DEBUG CONSOLE PROBLEMS

```
Post (Type)
+ Created type `Post`
+ Created field `id` of type `ID!`
+ Created field `title` of type `String!`
+ Created field `body` of type `String!`
+ Created field `published` of type `Boolean!`
+ Created field `author` of type `User!`
+ Created field `createdAt` of type `DateTime!`
+ Created field `updatedAt` of type `DateTime!`

User (Type)
+ Created field `posts` of type `[Post!]!`

PostToUser (Relation)
+ Created an inline relation between `Post` and `User` in the column `author` of table `Post`
```

Data Output

Explain

Messages

Notifications

	<div>id</div> <div>[PK] character varying (25)</div>	<div>name</div> <div>text</div>	<div>email</div> <div>text</div>	<div>createdAt</div> <div>timestamp without time zone</div>	<div>updatedAt</div> <div>timestamp without time zone</div>
1	ck6pbk8wp00190886pvflmb1m	John Doe	j.doe@email.com	2020-02-16 17:44:10.765	2020-02-16 17:44:10.765

Data Output

Explain

Messages

Notifications

	<div>id</div> <div>[PK] character varying (25)</div>	<div>title</div> <div>text</div>	<div>body</div> <div>text</div>	<div>published</div> <div>boolean</div>	<div>createdAt</div> <div>timestamp without time zone</div>	<div>updatedAt</div> <div>timestamp without time zone</div>	<div>author</div> <div>character varying (25)</div>
1	ck6pcod6s00pj0886klk4n7e0	Prisma Post		false	2020-02-16 18:15:22.549	2020-02-16 18:15:22.549	ck6pbk8wp00190886pvflmb...

Integrating Prisma into a Node.js Project

The big picture goal is to allow our Node.js application to read and write from the PostgreSQL database. Currently the Prisma GraphQL API can read and write from the PostgreSQL database and all we need to do is figure out how to allow Node to interact with this API. Once Node is able to interact with this API, Node will be able to read and write from the PostgreSQL database.

We need to perform some configuration to get this all working. First we would need the `node_modules`, `src`, `.babelrc`, `package.json` and `package-lock.json` files in our prisma project directory along with the generated prisma directory i.e. 6 folder/files in total which would make up our Node.js/Prisma GraphQL project directory.

We need to install two libraries into our project. The first is called `prisma-binding` which provides us bindings for Node.js i.e. a set of Node.js methods we can use that allow us to interact with our Prisma GraphQL API (<https://github.com/prisma-labs/prisma-binding>). This library is created by the creators of the Prisma itself and is a must have tool when working with Prisma and Node.js. To install this tool we would run the following terminal command within our project directory:

```
:~$ npm install prisma-binding
```

Once this is installed we would then need to create a file within the `src` directory called `prisma.js` and this file is where we would store all of the code required to connect our Node.js application to the Prisma

GraphQL API.

src/prisma.js:

```
import { Prisma } from 'prisma-binding';  
  
const prisma = new Prisma({  
  typeDefs: 'src/generated/prisma.graphql',  
  endpoint: 'http://localhost:4466/'  
});
```

We would import the Prisma named export which is a constructor function we can use to create a connection to a Prisma endpoint. We want to create a new instance of the Prisma constructor and store the value in a variable which we can call on. The Prisma constructor function takes in a single object argument. This is the options object where we configure Node.js to connect to the correct Prisma endpoint.

There are two things we have to provide in this object argument. The first is typeDefs i.e. the Type Definitions for the endpoint that we are connecting to. This is necessary for the prisma-binding library can generate all of the various methods needed. Therefore, it does not come with a createUser method unless we actually have a User Type that Prisma is going to interact with. If we have a Comments Type we are also going to have a .createComment method on prism.mutation and so on for all our other type definitions i.e. all of the things we see in GraphQL Playgrounds DOCS tab.

The second is the endpoint specified as a string which is the actual URL where the prisma GraphQL API lives.

The typeDefs is more complex to define compared to the endpoint and so we need to install a tool called graphql-cli (<https://github.com/Urigo/graphql-cli>). This is a CLI tool providing different commands to perform common tasks.

The datamodel.prisma file represents our data-model and it does not contain all of our Type Definitions. The Type Definitions is auto-generated by Prisma based off of the contents of datamodel.prisma. We are configuring the graphql-cli tool to fetch the auto-generated Type Definitions file which we can then reference that file by its path in our typeDefs object argument. This will then connect us to the Prisma API. The `graphql get-schema` command is the only command we would need to download the schema file from the endpoint.

To install this tool we would run the following command in the terminal:

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

Once this is installed we are going to setup a single new file into the project and setup a new script in package.json to run the `graphql get-schema` command. The file we create is going to be a configuration file telling the get-schema command where it can find the schema it should be fetching and where in the project it should be saving that file. The file must be called `.graphqlconfig` and live within the root of our project. This is a JSON configuration file where we provide the two pieces of information.

.graphqlconfig:

```
{ "projects": { "prisma": { "schemaPath": "src/generated/prisma.graphql", "extensions": {  
  "endpoints": { "default": "http://localhost:4466/" } } } } }
```

The common practice is to create a new directory folder in the src directory called generated. This is where we would store generated code like the file we are about to download. The schemaPath will provide the path to this folder and the file name which is typical to be named as prisma.graphql. This is where the Type Definitions will be saved and is also the path we would use in the prism.js typeDefs option argument.

We also have to specify the endpoint for the GraphQL Playground API which is localhost:4466 (unless you have a different IP address). We do this within the extensions property which allows us to define all sorts of things endpoint being one of them.

Finally we need to update the package.json file to add the graphql-cli command we would want to use to download the Prisma GraphQL API schema.

package.json:

```
{ ..., "scripts": { ..., "get-schema": "graphql get-schema -p prisma" } }
```

We can call the get-schema script whatever we would like and set its value to the `graphql get-schema` command followed by the `-p` flag and then the name of the project we specified in the `.graphqlconfig`. In the above example we only have one project which we named prisma. We can now run this script using the `npm run get-schema` command in the terminal to download and generate the schema file in the src/generated directory.

We would never edit or makes changes to this file and the only time it will change is when we run the script again to fetch the latest schema.

Therefore, if we make changes to the `datamodel.prisma` file we would deploy to Prisma GraphQL API and then fetch the updated schema via the `get-schema` script.

We now have everything configured and setup with our Prisma GraphQL API and our Node.js application so that Node.js can connect to the Prisma GraphQL API using the Type Definitions and endpoints. We can now look at how we can interact with the Prisma GraphQL API from inside of our Node.js application code.

Using Prisma-Bindings

Now that we know how to setup Prisma-Bindings into our Node.js projects we are now able to interact with our database from the server via our node.js code.

On the `prisma` object within the `prisma.js` file we have four key properties we can use. The `.query` property contains all of the queries we have access to, the `.mutation` contains all of the mutations we have access to, `.subscription` has all of the subscriptions we have access to and finally `.exists` which contains some handy utility methods.

The `.query` is nothing more than an object and on that object there are a set of methods available to us i.e. one method for every query the API supports. We access the query by name and we can see all of our queries within the DOCS tab within GraphQL Playground. Below is an example node.js (JavaScript) code for fetching all users:

src/prisma.js:

```
import { Prisma } from 'prisma-binding';  
  
const prisma = new Prisma({ typeDefs: 'src/generated/prisma.graphql', endpoint: 'http://localhost:4466/' });  
  
prisma.query.users(null, '{ id name email posts { id title } }').then((data) => {  
  console.log(JSON.stringify(data, undefined, 2));  
});
```

The method name matches exactly with the query name found in the GraphQL schema docs. We call this as a function and pass in the argument(s) the query accepts. All prisma methods takes in two arguments; the first are the operation arguments and the second is the selection set.

Not all methods require an operation argument and this may be optional as seen above. We would pass in null as the first argument if we are not passing in an operation argument. The second argument for the selection set is represented as a string. We open and close curly brackets and list out all of the things we want to receive back from the selection set.

What comes back from the function is a promise which means we would need to use `.then` and/or `.catch` to wait for the promise to resolve. This is an asynchronous operation i.e. it takes time to make the request to the GraphQL Prisma API and it takes time for Prisma to make a request to the underlying database. We pass in a callback function to get called when the promise resolves and this callback functions gets called with a single argument i.e. the data that came back from the query.

The data that comes back is exactly the same we would see when we execute and get back the query results in GraphQL Playground.

To make this file execute we need to make one change to our index.js file to import the prisma.js file:

```
src/index.js:
```

```
import './prisma';
```

We are not importing any named or default export and so we can leave off the exports list and the from keyword and directly reference the file path. This in effect will run the prisma.js file which is what we would want in order to execute the file and run our query to dump the results in the terminal.

We can run npm start in the terminal to get our index.js code running. We should see an array of users object containing the id, name and email properties printed to the terminal. We can change the selection set of the function argument to return exactly the data we would want to return back. The selection set is important because it determines the structure of the data that we get back from our queries/mutations/subscriptions exactly as we saw with GraphQL Playground.

We can use JSON.stringify to convert a JavaScript object/array into a JSON data which will allow us to customise how the data is displayed. If we did not use the JSON.stringify method this will output the data, however, the linked data posts would only display [Object] rather than the data in the console as hidden data. The .stringify method takes in three arguments which are the data object, the replacer function (which allows us to remove and replace properties) and finally the space to use when indenting the JSON data.

Therefore, we can use Prisma-Bindings in node.js to do anything we are able to do from GraphQL Playground but the advantage is that it runs completely inside of our code and we can use it to build out our app.

Mutations with Prisma-Bindings

The mutations property allows us to perform the create, update and delete operations from our node.js application. Just like `prisma.query` the `prisma.mutation` is an object with methods for every single mutation available to us from our schema docs. We can always go into GraphQL Playgrounds DOCS tab to view all the mutation available to us from our schema documentation. The arguments that get past into the mutation methods is exactly the same as the queries methods i.e. the operation argument and the selection types. With mutations we typically need to pass in operation arguments, below is an example code:

[src/prisma.js:](#)

```
import { Prisma } from 'prisma-binding';

const prisma = new Prisma({ typeDefs: 'src/generated/prisma.graphql', endpoint: 'http://localhost:4466/' });

prisma.mutation.createPost({
  data: { title: "New Post", body: "New Post Body Text" published: true, author: { connect: { id: "123abc" } } } }, { id title body published }
}).then((data) => { console.log(data); });
```

The first argument is an object and this object is where we provide all of the operation argument. This again returns a promise which we can run a callback function passing in a single argument of the return data which we can display in the terminal. Since we are not returning a linked data we do not need to use `JSON.stringify` to display hidden objects in the terminal output.

We can actually chain on some of the calls together to do something before or after we do so something else. We can use promise chaining to get this done. Below is an example:

src/prisma.js:

```
import { Prisma } from 'prisma-binding';

const prisma = new Prisma({ typeDefs: 'src/generated/prisma.graphql', endpoint: 'http://localhost:4466/' });

prisma.mutation.createPost({ ... }).then((data) => {
  console.log(data);
  return prisma.query.users(null, '{ id name posts { id title } }');
}).then((data) => {
  console.log(JSON.stringify(data, undefined, 2));
});
```

We would use the return keyword to return another chained promise which would allow us to call on another .then method which would fire when our second promise resolves. The above will now execute the code to create a new post and once it resolves it will print the return data in the terminal and then the second promise of the query method to fetch all users. When the second promise resolves this will print the returned data to the terminal. This would now include the newly created post. To conclude, using mutations inside of node.js is exactly the same as using mutations in GraphQL Playground.

Using Async/Await with Prisma-Bindings

Since all of the prisma-bindings methods return back promises we are able to utilise JavaScripts async/await syntax where promises are used. You should be familiar with how the async/await syntax works as this is a basic JavaScript feature. Below is an example code using the async/await with prisma-bindings:

src/prisma.js:

```
import { Prisma } from 'prisma-binding';

const prisma = new Prisma({ typeDefs: 'src/generated/prisma.graphql', endpoint: 'http://localhost:4466/' });

const createPostForUser = async (authorId, data) => {
  const post = await prisma.mutation.createPost({
    data: {
      ...data,
      author: { connect: { id: authorId } }
    }
  }, '{ id }');

  const user = await prisma.query.user({ where: { id: authorId } }, '{ id name email posts { id title published } }');

  return user;
};
```

```
createPostForUser('123abc', {  
  title: "A book to read", body: "The War of Art", published: true  
}).then((user) => { console.log(JSON.stringify(user, undefined, 2) });
```

We would setup a const variable to store a function (using either a ES5 function or a ES6 arrow function syntax). Before the argument list of the function we would use the async keyword to mark the function as an asynchronous function. The argument list above contains all the information i.e. the author the post should be associated with and the post information stored as an object. Within the function body we can go through the process of creating a post using prisma-bindings.

We have a const post variable which will store the post data that comes back from calling createPost method. We would set its value to the await keyword to await for the promise to resolve to get the resolved value i.e. this replaces the .then syntax.

We use the spread operator (...) to spread out all of the data object (argument) properties for the post i.e. the title, body and published properties that live on the data argument. We can also add to the data object to pass in the author id which would be stored in the authorID argument. Finally we would provide the selection type to grab the information from the newly created post.

The second statement chains on after the first async function resolves to fetch the user. Once again we would create a variable to store the async/await function for the user query. In the above example we return back from the selection type argument the id, name and email of the author and also the posts id title and published values when the asynchronous function resolves.

We can now do something with the information that we have compiled and return something back from this overall function such as the user data. We can now call on this function passing in the relevant arguments to create a new post and return the user data from this prisma-binding.

We must remember that async functions always return a promise and the resolved value of that promise is whatever we return from the function. We can therefore add a single `.then` call on the function to do something with the returned value, in the above example the user value is printed to the terminal.

To conclude, we now have a way to use `async/await` with prisma-bindings allowing us to vastly simplify the asynchronous code and we are still able to easily chain on multiple asynchronous code to return a promise and then do something with that resolved promise value.

Checking If Data Exists Using Prisma-Bindings

The `prisma.exists` property comes with some handy utility functions that make it easy for us to determine if there is a record of a given type for example, we can check if a given user exists. This allows us to perform a check to verify the data exists before we perform mutations.

The `.exists` has a dynamic set of methods available to us similar to the `.query` and `.mutation` property. We have one method for every single type and the method name is the type name. The `.exists` method takes in a single argument which is an object. On this object we provide all of the properties

we would want to verify about the data we are looking for. So in the below example, if we want to verify whether a User exists with a given id we would set the id property followed by the id value we are searching for:

src/prisma.js:

...

```
prisma.exists.User({ id: "abc123" }).then((exists) => { console.log(exists) });
```

All `.exists` methods return a promise and that promise resolves to a boolean value of true or false i.e. the data we are looking for either exists or it doesn't.

We can perform more complex assertions and it is up to you as the developer to set as much criteria(s) actually needed although typically we would check for the id for the record we are looking for. We can also provide associated assertions for example we can search for a comment by an id and that it was created by the following author:

```
prisma.exists.User({ id: "abc123", author: { id: "123abc" } }).then((exists) => { console.log(exists) });
```

The above example would check if the comment of the id exists and that the author is also the id we asserted and would return either a true or false resolved promise value. Therefore, we can make the assertion as simple or complex as we need. We can chain on the promise when it returns true to then run our query or mutations. We can use the `throw new Error` to throw an error if the returned boolean is false using an if statement to check the promise value. Below is an example code using the example from the last section to demonstrate:

src/prisma.js:

...

```
const createPostForUser = async (authorId, data) => {  
  const userExists = await prisma.exists.User({ id: authorId });  
  if(!userExists) { throw new Error('User not found'); };  
  const post = await prisma.mutation.createPost({  
    data: {  
      ...data,  
      author: { connect: { id: authorId } }  
    }  
  }, '{ author: { id name email posts { id title published } } }');  
  return post.author;  
};  
  
createPostForUser('123abc', {  
  title: "A book to read", body: "The War of Art", published: true  
}).then((user) => { console.log(JSON.stringify(user, undefined, 2) }).catch((error) => {  
  console.log(error.message);  
});
```

This small modification to how we call the createPostForUser allows us to use a .catch method on our
100

function call to fire whenever we get any errors. The `.catch` method gets called with a function that has a single argument of the error object. We can use this to display the error in the console. Using `error.message` will limit the display of the error message property only, which is where the error string message exists i.e. a simplified error message displayed.

We can now make more robust prisma-bindings to check for data to exists before we blindly pass in data along assuming the data (e.g. user, post or comment) exists.

Customising Type Relationships

The screenshot shows the GraphQL Playground interface. The top bar contains tabs for various queries and mutations: `users` (Q), `createUser` (M), `updateUser` (M), `deleteUser` (M), `createPost` (M), `updatePost` (M), `createComment` (M), `comments` (Q), and `deleteUser` (M). The URL bar shows `http://localhost:4466/`. The left pane contains the following query:

```
1 mutation {  
2   deleteUser (  
3     where: {  
4       id: "ck6pe2o3p01aa0886o04f1xwd"  
5     }  
6   ) {  
7     id  
8     name  
9     email  
10  }  
11 }
```

The right pane shows the JSON response:

```
{  
  "data": {  
    "deleteUser": null  
  },  
  "errors": [  
    {  
      "locations": [  
        {  
          "line": 2,  
          "column": 3  
        }  
      ],  
      "path": [  
        "deleteUser"  
      ],  
      "code": 3042,  
      "message": "The change you are trying to make would violate the  
required relation 'CommentToUser' between Comment and User",  
      "requestId": "local:ck6zk995r004t0886proc1l9d"  
    }  
  ]  
}
```

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

We would notice that when we try to delete certain data from our database as seen in the above example we would not get the expected results. Instead we would see an error message of “The change you are trying to make would violate the required relation ... between ... and ...” — this would inform us of the Type relationships found in our `datamodel.prisma` file.

In the above example the deletion of the User would make all of the Comments invalid because the comments require an author field which needs a link to a user and this is a non-nullable Type. This is why Prisma is not allowing us to delete the user.

We can overwrite this behaviour by setting some on-delete behaviour to solve this issue. The process of setting this up is very simple. Prisma provides us two options: the default `SET_NULL` option and the `CASCADE` option. The `CASCADE` option allows us to overwrite the default `SET_NULL` behaviour. These options become relevant when we have types that link to other types.

We can customise what happens to the other records when a given record gets removed. So when we try to delete the user the related posts and comments fields are linked to the Post and Comment Types which reference the User Type as a non-nullable field. Therefore, the default `SET_NULL` cannot apply null value to the author field and this is the cause of the error.

We would generally want to delete the posts and comments when a user is deleted and therefore the default `SET_NULL` option is not what we are looking for. In these cases we are looking for the `CASCADE` option. This will delete all posts and comments associated with a deleted user. We can also set this for when a post gets deleted all of the associated comments also get deleted.

To set this up Prisma offers us a directive called `@relation` which we can use on fields that link to other types.

`prisma/datamodel.prisma:`

```
type User { ..., posts: [Post!]! @relation(name: "PostToUser", onDelete: CASCADE) }
```

```
type Post { ..., author: User! @relation(name: "PostToUser", onDelete: SET_NULL) }
```

The `@relation` takes in an argument list of two properties. The first is the name which contains the name of the relation and the second is `onDelete` which holds the delete option between `SET_NULL` or `CASCADE`.

We should always use a descriptive name for the relationship. In the above example we use the `@relationship` on the User Type to cascade the deletion of posts when a user is deleted. We also setup the `@relation` directive on the Post Type; however, the `onDelete` is set to the `SET_NULL` option because we would not want to cascade the deletion of the user when a post is deleted. We would need to matchup the name of the relationship on the other end i.e. the User and Post type must have the same name value to link each relation.

When making changes to the `datamodel.prisma` file we would need to ensure that we re-deploy/push the changes to our prisma server by running the `prisma deploy` command in the terminal within our prisma directory. Once the changes have been deployed we are now able to run successfully the exact same delete command as seen in the GraphQL Playground screenshot example above without running into any errors.

To conclude prisma provides us a very simple way to delete relation data using the `@relation` directive.

Section 5: Authentication with GraphQL

Adding Prisma into GraphQL

We would want to add Node.js between our public users and the private Prisma GraphQL API to prevent users from directly interacting with the Prisma GraphQL API. If the user had direct access to the Prisma GraphQL API this would allow them to do whatever they want with the data in the database. Instead, we would want Node.js to act as the middleware/middleman to provide our application with Authentication and data validation.

There are certain data which make sense to expose the data even when someone is not authenticated such as published posts while there are other data which should not be exposed such as a user's draft post.

To setup Node.js between the public user and the Prisma GraphQL API we would first start with our `prisma.js` file. We would need to add an export so that other files can access the prisma object to use the prisma object's properties and methods available such as `.query`, `.mutations`, `.subscription` and `.exists` methods.

`src/prisma.js`:

```
import { Prisma } from 'prisma-binding';  
const prisma = new Prisma({ ... });  
export { prisma as default };
```

Next we would need to setup the prisma to live in our application context object within the index.js file.

src/index.js:

```
import prisma from './prisma';  
const server = new GraphQLServer({ ..., context: { db, pubsub, prisma } });
```

We can now go into our resolvers directory and take advantage of prisma which is now part of the application context within our .js files. Below is an example of the Query.js file taking advantage of the prisma object:

src/resolvers/Query.js:

```
const Query = { users(parent, args, { prisma }, info ) { return prisma.query.users(null, info ) } };
```

All prisma methods takes in two arguments, the operation argument and the selection set. The selection set argument can take in three distinct type of values i.e nothing, a string or an object. There are situations where we would select one value type over the other.

Firstly, we can either leave the second value off or set a falsey value such as null or undefined. In this case prisma will fallback to a default which would grab all of the scalar fields for the Type. The problem with providing nothing as the value type is that we can never ask for relational data.

The second value type is a string value for the selection set which we have already explored in the last section. The problem with providing a string is that we have to explicitly define what we need as a string value and sometimes we may not know what we need.

Finally, we can provide an object. This object is not defined by us but one that is created for us i.e. the info object. The info object contains all of the information about the original operation i.e. the operation from a web browser or an iOS app. This gets sent to the server and is then accessible to prisma via the info object. This applies to all resolvers whether they are queries, mutations or subscriptions. Therefore, the user can provide the selection types to return when they perform the operation on their browser or app which is then accessible from Node.js. This includes access to relational fields.

The resolver methods can take a promise as the return value i.e. it will wait for the promise to resolve and then it will send that data back. This mean we can return the prisma method as seen in the above example.

We have now explored the process of setting up Node.js between the public API and private Prisma GraphQL API. For the moment there are no restrictions whatsoever and the users have access to the database.

Integrating Operation Arguments

To integrate operation arguments to our Prisma methods we can create an object variable and pass this object as the first argument. If the object is empty this is the same as passing in null as the first argument. We can setup a series of conditional statements to make changes to the object variable overtime. Below is an example of looking at the args argument to modify the object:

src/resolvers/Query.js:

```
const Query = {  
  const opArgs = { };  
  users(parent, args, { prisma }, info ) {  
    if(args.query) { opArgs.where = { name_contains: args.query }; };  
    return prisma.query.users(opArgs, info );  
  }  
};
```

The code block runs only if the args.query exists. This will make changes to the structure of the opArgs object variable which would then get passed in as the operation argument for the prisma method. We would need to look at the GraphQL schema to see what query arguments the operation can take in order to change the object correctly.

We can now run the query for users in the GraphQL Playground API (i.e. on <http://localhost:4000/>). If a query argument is not provided the user query will continue to work. However, if a query is provided as an operation argument then the Node.js query operation will also work and will return any users which contains the query string.

This is demonstrated with the two example screenshots below of both scenario within GraphQL Playground public API.

Q users ×

Q posts

+

PRETTIFYHISTORY

● http://localhost:4000/

COPY CURL

1 query {
2 users {
3 id
4 name
5 email
6 }
7 }

▶

{
 "data": {
 "users": [
 {
 "id": "ck6pbk8wp00190886pvflmb1m",
 "name": "John Doe",
 "email": "j.doe@email.com"
 }
]
 }
}

DOCS

SCHEMA

Q users ×

Q posts

+

PRETTIFYHISTORY

● http://localhost:4000/

COPY CURL

1 query {
2 users(
3 query: "ohn"
4) {
5 id
6 name
7 email
8 }
9 }

▶

{
 "data": {
 "users": [
 {
 "id": "ck6pbk8wp00190886pvflmb1m",
 "name": "John Doe",
 "email": "j.doe@email.com"
 }
]
 }
}

DOCS

SCHEMA

There are all sorts of ways we can filter our data which can be found in the GraphQL Playground DOCS tab for the query.

Custom Type Resolvers

We can leave the Custom Type resolver objects empty but leaving it in place and can run a query in the GraphQL Playground API to return back the relational data.

src/resolvers/Users.js:

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

This will continue to work by default in prisma because when we setup the info argument as the second argument to the prisma method, the prisma method now has access to the selection set from the fields selected from the user operation. This includes not only the scalar fields but also the relation fields. Therefore, there is no need to add more code. This is another great reason to use Prisma because it works directly with GraphQL and makes it really easy to setup the Node.js application.

The alternative method is to add prisma to the Custom Type and query the relation data as seen in the last two sections; however, compared to the above approach this is more code added to the application which is unnecessary bloat. Now that we know how to implement Prisma into our Node.js application we can now focus on adding Authentication with Node.js.

Prisma and Subscriptions

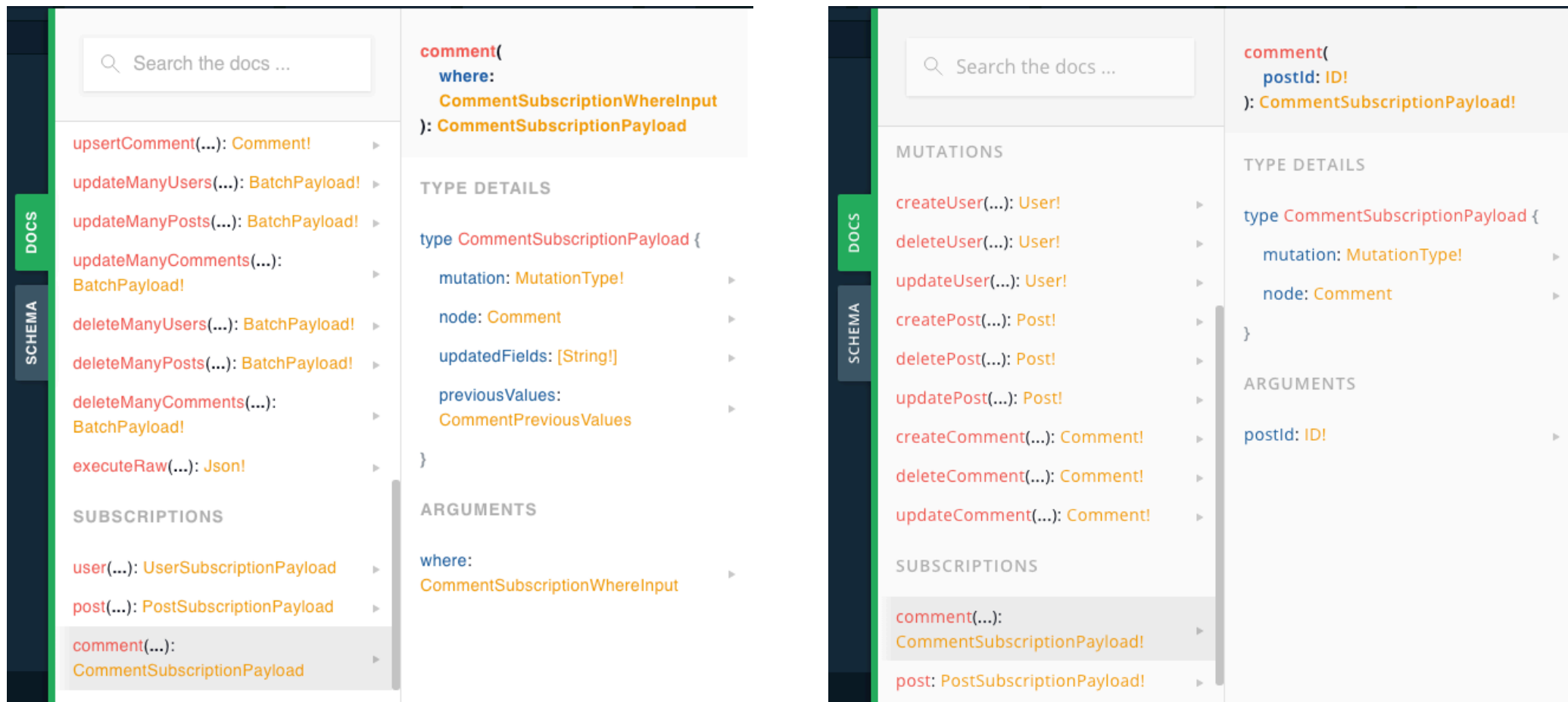
Prisma has built-in support for subscriptions and the setup process is very simple to implement without the need for pubsub.publish calls. With Prisma all of this occurs behind the scenes.

When data flows within our application it flows from Prisma to Node.js and then from Node.js to the Client (GraphQL Playground). The same process will occur with the subscriptions; however, we would need to ensure the data that Node is sending to the Client aligns with the data Prisma is sending Node. If they do not align this would mean we would lose data in the application.

In our examples if we look at the Prisma schema documentation on <http://localhost:44666> we would see that the subscription returns the selection types of mutation, node, updateFields and previousValues. If we compare this with our <http://localhost:4000> Schema documentation (i.e. the Client) we would notice only two fields are sent back from the subscription of mutation and data.

Therefore, if Prisma send all the fields to Node.js but Node.js sends different fields to the Client this would mean not all of the data is getting to the Client. We would also notice that there are no data field coming back from Prisma and therefore things would fail because the data field is non-nullable.

As a result Node.js does not serve as a good middleware between Prisma and the Client. What we would need to do to fix this is to align the selection types returned from the Prisma subscription with the type definition for our Client. In the example the only data that would be sent successfully to the Client would be the mutation selection type as it aligns with both Prisma to Node.js and Node.js to the Client.



To fix this issue we would make a small change to the schema.graphql file:

src/schema.graphql:

```
type CommentSubscriptionPayload {
  mutation: MutationType!
  node: Comment
}
```

This small change would align the Node.js to Client subscription with the Prisma to Node.js subscription. We can therefore integrate Prisma into our application to use the .subscription property methods as demonstrated below:

src/resolvers/Subscription.js:

```
const Subscription = {  
  comment: { subscribe(parent, { postId }, { prisma }, info) {  
    return prisma.subscription.comment({  
      where: { node: { post: { id: postId } } }  
    }, info);  
  }  
};
```

As we can see Prisma makes it very simple to add subscriptions to our application with very few lines of code compared to what we had previously with pubsub.published.

We now know how to place Node.js between our Client and Prisma GraphQL API. This is now going to allow us to authenticate and validate requests before actually reading and writing to the database.

Closing Prisma to the Outside World

We can use our Node.js server to check if someone actually has permission to read or write some data before the operation is actually performed. The problem with the current setup is that this is not enforceable because the Client can directly communicate with Prisma i.e. the user can go to <http://localhost:4466> and play around with the database all they want using the Prisma GraphQL Playground i.e. there is no need to go through Node.js GraphQL API. Therefore, we need to cut off

this channel for the Client and force the Client to use the Node.js channel.



To do this we setup what is known as the Prisma secret which is nothing more than a password. This password is required to communicate with Prisma. We would set this up on the Node.js and Prisma backend so that when Prisma asks what is the secret, Node.js would be able to provide the secret response correctly. At the end of the day any Client in the world would be able to interact with our Node.js API which we can then authenticate and validate those requests before we ever touch the database. This is the strategy we are going to use to lock down our data.

To set this up we would need to update two files which are the `prisma/prisma.yml` and the `src/prisma.js` files. These are the two files for both Prisma and Node.js servers backend. Below is the example code for setting this up.

`prisma/prisma.yml`:

```
endpoint: http://localhost:4466
```

```
datamode: datamodel.prisma
```

```
secret: supersecretpassword
```


The name of the 'secret' property is important and must be spelt correctly and the value can be set to anything we would like. We can then redeploy the file to Prisma by navigating to the prisma directory in terminal and running the `prisma deploy` command. This will now lock down prisma and anyone who wants to communicate with it would need to provide the secret password.



The screenshot shows the GraphQL Playground interface. At the top, there are tabs for various queries and mutations: 'users', 'comments', 'createUser', 'updateUser', 'deleteUser', 'createPost', 'updatePost', 'createComment', and 'deleteUser'. Below these tabs is a 'PRETTIFY' button, a 'HISTORY' tab, and a URL bar showing 'http://localhost:4466/'. 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 GraphQL query:

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

 The right panel shows the response, which is an error:

```
{
  "errors": [
    {
      "message": "Your token is invalid. It might have expired or you might be using a token from a different project.",
      "code": 3015,
      "requestId": "local:ck794cgod00200786f103pkpx"
    }
  ]
}
```

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

We would now see the error message when we now make any requests to our Prisma GraphQL API regardless if we try to connect to Prisma via our Node.js/Client GraphQL API or directly with the Prisma GraphQL API via GraphQL Playground.

Information: the endpoint in the prisma.yml file of `http://localhost:4466` is the same as the path `http://localhost:4466/default/default` and which is why we see `default$default` in PGAdmin schema for the database. We can change this endpoint to something like `http://localhost:4466/reviews/default` and this would create a new path location for the Prisma GraphQL API and we would see in PGAdmin `reviews$default` within the schema for the database.

We can provide the same secret to our prisma.js file as seen below:

src/prisma.js:

```
... const prisma = new Prisma({  
  typeDefs: '...', endpoint: '...', secret: 'supersecretpassword'  
});
```

In the prisma object variable we need to provide a property called 'secret' (again the name of the property is important and must be spelt correctly) and set its value the same as the value setup in the prisma.yml file. We are now able to communicate with Prisma via Node.js.

If we now try to perform a query in GraphQL Playground on <http://localhost:4466> we should continue to see the error message as seen in the screenshot above; however, if we try to perform a query in GraphQL Playground on <http://localhost:4000> this should successfully run without any errors because it is queried using our Node.js server.

We now have a way to force all communications to go through Node.js and locked any communication from the outside world directly to our Prisma GraphQL. This is an important first step to setting up things such as authentication and validation via our Node.js middleware.

If we want to continue to use the <http://localhost:4466> GraphQL Playground API for development we do this securely by setting up an authorisation token which is done through the HTTP HEADERS.

This is nothing more than a JSON object with key:value pairs. The key is the Header name and value is the Header Value. Below is an example of setup:

QUERY VARIABLES HTTP HEADERS (1)

```
1 {  
2   "Authorization": "Bearer eyJhbGciOiJIUzVCJ9.eyJkYXRhIjp7InNlcnZpY2UiOiJk.AgqfbbLh-iIL31KmBkTVAvqHYlhZUU10swERl+BDDSQ"  
3 }
```

The Header Key is “Authorisation” and its value is “Bearer ” followed by the token. The token we have to provide is not the secret but actually a token Prisma can generate for us and we generate this by running a single command. We need to go to the terminal within our prisma directory and run the following command to generate the token:

```
:~/prisma$ prisma token
```

This will generate a authorisation token that we can use for standard HTTP request which we can add inside of a Header allowing us to authenticate that we do have access to the Prisma GraphQL API. It is important to provide a space between the Bearer and token for this to work correctly.

We can now access and fire off requests securely within our http://localhost:4466 GraphQL Playground API without running into any error message as we did before but have it closed and secured against the outside world. We can now continue on to the next step of authentication/validation via the Node.js server.

Allowing for Generated Schemas

We now have a secret which closes the back channel from the Client to Prisma forcing the Client to go through the Node.js server in order to read/write from the database. We are now going to focus on how we can authenticate a Client's request.

A user would need to login to the application by providing something like an email/username and password credentials to the Node.js server. Node.js will verify that the user exists in the database with the credentials provided and if it finds a user matching the credentials it will create and send back an auth token to that Client. The Client can then use this token to make requests that require authentication for example editing a post.

When a Client now makes future requests to the Node.js server, it can provide the auth token with its request and Node.js can verify that the token actually belongs to the user and if it is a match it can actually perform the operation to the database via Prisma GraphQL API. The Client can use this token to make other requests which require an auth token.

This is the authentication mechanism we can setup to protect and lock down our GraphQL API operations from any unauthorised or unwanted requests providing a much more secure API.

The first step is to add a new field for the User Type to store the user's password alongside their name and email field. We would not store the password as plain text but rather use a hashing algorithm to securely store user passwords.

```
prisma/datamodel.prisma:
```

```
type User { id: ..., name: ..., email: ..., password: String! }
```

```
src/schema.graphql:
```

```
type User { id: ..., name: ..., email: ..., password: String! }
```

We would update both the User Type model on our Prisma and Node.js files to include the password field of a non-nullable String scalar type i.e. the Client and Prisma schemas should mirror each other to allow Node.js to communicate successfully between the two as the middleware.

We would need to wipe the database and re-deploy Prisma as we have now setup a new non-nullable field to the database. There is a shortcut way of performing this via a terminal command within the prisma directory path:

```
:~/prisma$ prisma delete
```

```
:~/prisma$ prisma deploy
```

The prisma delete will delete the default\$default service i.e. wipe all of our data in the database. We would need to answer y to the question to delete the database. We can then redeploy prisma to recreate the database so that the each and every User now has a non-nullable password field.

Finally we need to run the following command to get a new generated file of our src/generated/prisma.graphql containing the new schema.

```
:~$ npm run get-schema
```

We would run this command within our root project i.e. graphql-prisma directory. However, by setting up the prisma secret we have now locked out this command from being able to do its job correctly. There are a couple of ways we can overcome this problem such as adding a whole bunch of arguments to the script. The alternative is to setup a small tweak to the .graphqlconfig file:

.graphqlconfig:

```
{ "projects": { "prisma": { ..., "extensions": { "prisma": "prisma/prisma.yml", "endpoint": { ... } } } } }
```

The only reason this works is because we have the prisma directory in the same project directory and therefore we point the get-schema towards the prisma.yml file located in the prisma directory. This will allow it to get the schema without needing to go to http://localhost:4466 which is locked down. We do this by adding a single property to the extensions object which we call prisma and set its value to be directed to the prisma.yml file path.

This continues to lock down Prisma GraphQL API from the Client but we have provided an alternative for our local development tools allowing us to keep the script exactly the same without adding any additional arguments. We can therefore run the script without running into any errors and have an updated generated schema.

We can now focus on making changes to the Node.js Mutations.js file by updating the createUser

mutation to perform some validations and hash the password without storing any plain text passwords. We will explore this in the next section.

Storing Passwords

In the `src/schema.graphql` file we have the `createUser` mutation which takes in the `CreateUserInput` type for the data. This currently takes in the name and email. We would need to update this to add the password as an input field for the method.

`src/schema.graphql:`

```
type Mutation { createUser(data: CreateUserInput!): User! }  
type CreateUserInput { name: String!, email: String!, password: String! }
```

Whenever someone tries to run the `createUser` mutation they will have to provide a password. We can now perform some validation on the password for example the length to make sure it is at least 8 characters long. To do this we would update the `src/resolvers/Mutation.js` file:

`src/resolvers/Mutation.js:`

```
const Mutation = { async createUser(parent, args, { prisma }, info) {  
  if(args.data.password.length < 8) { throw new Error('Password must be 8 characters or  
  longer') };  
  return prisma.mutation.createUser({ data: args.data }, info);  
} };
```


We would use a if statement to check the length of the password and throw an error if it does not pass the validation.

Now that we can validate the password we should store the password in the database with the new user; however, it is never a good idea to store the password as plain text. We would pass the plain text password through a hashing algorithm. There are many hashing algorithm some are insecure while others are secure and great for production use. A very popular and secure hashing algorithm is the bcrypt algorithm which we can install the npm package and use its methods in our projects (<https://www.npmjs.com/package/bcryptjs>). To install this package we can run the following command:

```
:~$ npm install bcryptjs
```

We can now import this module and use the .hash method to hash the password before storing hash version of the password in the database:

[src/resolvers/Mutation.js:](#)

```
import bcrypt from 'bcryptjs';

const Mutation = { async createUser(parent, args, { prisma }, info) {
  if(args.data.password.length < 8) { throw new Error('Password must be 8 characters or...') };
  const password = await bcrypt.hash(args.data.password, 10);
  return prisma.mutation.createUser({
    data: { ...args.data, password }
  })
}
```

```
}, info);
```

```
} };
```

Hash passwords are one way which means we can take a plain text password and hash it but we cannot take the hash version and get the plain text version back which is a good thing.

The `.hash` method takes in two arguments the first is the plain text password and the second is the salt which is a numeric value of the length we want to use. A salt is nothing more than a random series of characters that are hashed along with the string we are hashing. This makes the hashing algorithm more secure which would randomise the plain text each time its run through the `.hash` method.

The `.hash` returns a promise and the promise resolves with the hashed value i.e. a hashed string. We can store this hashed value in a variable. We can create a new object using the spread (...) operator to spread out all of the properties on `args.data` and then set the password property to the hashed password stored in the password variable. Where we setup an object property where its value comes from a variable of a same name (i.e. `password: password`) we can use the ES6 shorthand syntax. This will override the plain text password with the hashed version which would then be saved to the database using the `prisma.mutation.property` method.

We now have a secure way to store passwords securely to our database against the created users. The next step is to be able to generate and send back a auth token whenever a new user signs up or an existing user signs into the application.

Creating Auth Tokens with JSON Web Tokens

The last step is to create and send back an authentication token which the Client stores and passes along with future requests whenever the Client wants to do something that requires authentication for example creating a post or deleting a comment.

There are plenty of different ways of creating authentication tokens and in this guide we are going to look at the very popular approach of the JSON Web Token (JWT). This is an open standard which gives us a way to securely transmit information between parties and also tamper proof. Each token is associated with a specific user and will expire after a certain amount of time.

To use JWT in our application we can install a library called jsonwebtoken (<https://www.npmjs.com/package/jsonwebtoken>) which will help implement the JWT integration. To install this package we can run the following terminal command within our project directory:

```
:~$ npm install jsonwebtoken
```

After installing the package we can import the token into our JavaScript files.

src/resolvers/Mutation.js:

```
import jwt from 'jsonwebtoken';
```

There are a few different methods in this library we can use and will be demonstrated below to

analyse how jsonwebtoken works in practice to understand how we can implement authentication using JSON Web Tokens in our application API.

The first method `.sign` allows us to create a new token. A token is nothing more than a really long string. This method requires two arguments the first is an object which is known as the payload and is used to provide information for our specific purposes. It is important to note we can put whatever we want into this object i.e. we can add as many properties as we need and in the below example we use an id to associate the token with a user (note we could have named id: something different). The second argument is the secret which is used to verify the integrity of the token making sure the token is not tampered with for example changing the id to a different user. The secret is something that is only going to live on the Node.js server.

What comes back from the `.sign` method is the tokens which we can save to a variable and do something with it. We can `console.log()` to view the output token of the `.sign` method in the terminal.

```
const token = jwt.sign({ id: 1 }, 'nodesecrettokenpassword');  
console.log(token);
```

We would see a very long token in the terminal which is what the server is going to send to the Client whenever the Client signs up or logs in to the application. This would be the same token the Client would store and send along with future requests to our Node.js server whenever it wants to be authenticated. It is important to note that the payload is not meant to be encrypted and should be publicly readable which is OK because this is all part of the JWT standard.

The next method `.decode` allows us to decode a token. This takes in one argument which is the string token and does not take in a secret. This returns back to use the decoded payload which is an object with the original payload setup properties and an additional `iat` (issued at time) property which stores a timestamp for when the token was initially created. The `iat` is a useful built in feature within the JWT.

Again, this demonstrates that the payload is not meant to be encrypted under the JWT standard.

```
const decoded = jwt.decode(token);
```

```
console.log(decoded);
```

Example Output: { id: 1, iat: 1533467675 }

The final `.verify` method verifies that the token was actually created from the server. This prevents the Client spoofing the a token and sending it to the server. This method decodes the token but also verifies that the token was indeed created with a specific secret. This will ensure that the tokens that we are reading are tokens created by the Node.js server and not by anyone else. This method takes in two arguments the first is the token we want to verify and the second is the secret. If the token was not created by the same secret then verify will fail. Only when there is a match will verify succeed.

This returns back decoded data which we can create a variable and store the value.

```
const decoded = jwt.verify(token, 'nodesecrettokenpassword');
```

```
console.log(decoded);
```

Example Output: { id: 1, iat: 1533467675 }

What we would see in the terminal is again similar to the `.decoded` method which is an object holding the payload properties and a `iat` property; however, this is a decoded and verified data. It is important to note that we would in practice use the `.verify` method over the `.decoded` method on our server.

When a Client tries to spoof a token using a secret that does not match servers secret, this will throw an invalid signature error in the terminal to let us know that the token that was trying to verify could not be verified and therefore cannot be trusted. This ensures that the Client cannot tamper with the token and it all comes down to this secret which the Client will never know.

When we use the `.sign` method to generate a token, the token generated is not a totally random token. We can visit the following website to analyse the token: <https://jwt.io/>

We would notice the token has three distinct sections separated by a period (.) character. The first section is the Header data, the second section is the payload data and the last section is the verify signature.

The Header contains information about the token in general and is not something we the end user will ever going to modify or read from. This is used for internal purposes only i.e. the type which is JWT and the algorithm used to generate the signature e.g HS256 algorithm. The Payload is the data we would see in the terminal when we use either `.decode` or `.verify` methods.

The Verify Signature for the token which keeps everything tamper proof and secure. The signature is nothing more than a hash i.e. we hash the Header with the payload together with our secret which results in the Verify Signature we see in the token final section.

Both the .sign and .verify methods actually generate a Verify Signature which is stored in the third section. When we create a token it takes the header and payload data and hashes it with the secret. The same occurs when we verify a token which takes the header and payload and hashes it with the passed in secret. If we get the same Verify Signature this shows that the token is valid and if it does not match this indicates that a different secret was used and is not valid. This ensures the token is secure and tamper proof.

While the Header and Payload sections of the token may look encrypted it is actually not encrypted but are Base64 encoded JSON.

Now that we are aware of the inner workings of the JSON Web Tokens we should be comfortable in using the library in our Mutations.js file within to generate and send back auth tokens whenever a user signs up or logs in to the application. Below is an example for the createUser mutation method:

src/resolvers/Mutation.js:

```
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';

const Mutation = { async createUser(parent, args, { prisma }, info) {
  const password = await bcrypt.hash(args.data.password, 10);
  const user = prisma.mutation.createUser({ data: { ...args.data, password } });
  return { user, token: jwt.sign({ userId: user.id }, 'nodesecret') }
} };
```


We now have everything setup to return a object that contains two things which is the user data returned from the prisma mutation and the generated token. The final change is to the return value of the createUser method in the schema.graphql file so that the mutation returns a type that has both the User and token on it:

src/schema.graphql:

```
type Mutation { createUser(data: CreateUserInput!): AuthPayload! }
```

```
type AuthPayload { token: String!, user: User! }
```

```
type CreateUserInput { name: String!, email: String!, password: String! }
```

The value we expect to come back from the createUser mutation method matches with the value we are returning from the method and this will prevent any errors being thrown.

Notice that we have left off the info argument off as the second argument to createUser mutation because this would throw an error. This is because we cannot select a token or a user on what comes back from the createUser mutation. We can only select the fields from the user. Therefore, in this case where we are returning something custom such as the object containing user and token, using info is actually not what we want. Instead we would leave off the second argument altogether in the prisma.mutation.createUser which would then return all scalar fields. We would now have a successful mutation for creating new users that would return both the created user's details as well as the generated auth token that the Client can store and use later to perform privileged operations for example creating a new post or updating a comment.

We now have one side of the authentication setup completed i.e. we know how to generate tokens but we do not know how to actually verify a token and use the token to determine whether or not a Client is able to do something. This is something we will explore in following sections.

Logging in Existing Users

Now that we know how to hash passwords and store it in a database we now need to learn how to log in a user who would provide their plain text password. This requires us to perform a comparison of the plain text password with the hashed password in the database to check for a match.

To compare the two passwords the bcrypt library provides a method called `.compare` which takes in two arguments. The first is the plain text password and the second is the hashedPassword. This returns a promise which resolves to true if the plain text password is a match with the hashed password else it would return false otherwise. Below is an example syntax:

```
const isMatch = await bcrypt.compare(password, hashedPassword);  
console.log(isMatch);
```

Usually with a login system a user would have to provide their username/email and their plain text password. We can retrieve the hashed password from the database that is associated with the username/email and then use the `.compare` method to compare the plain text password provided with the hashed password retrieved from the database. If this returns true we can log in a user and provide an Auth Token to the Client.

It is important to note that the bcrypt algorithm is a one way hashing algorithm and therefore are not decrypting the hashed password instead we are hashing the plain text password and comparing it to the database hashed password. There is no way to get the original plain text value from the hashed password.

Below is an example of setting up a login mutation:

src/schema.graphql:

```
type Mutation { login(data: LoginUserInput!): AuthPayload! }  
type AuthPayload { token: String!, user: User! }  
input LoginUserInput { email: String!, password: String! }  
type User { id: ID!, name: String!, email: String!, password: String! }
```

src/resolvers/Mutation.js:

```
const Mutation = {  
  async login(parent, args, { prisma }, info) {  
    const user = await prisma.query.user({ where: { email: args.data.email } });  
  }  
  if(!user) { throw new Error('Unable to login'); };  
  const isMatch = await bcrypt.compare(args.data.password, user.password);  
  if(!isMatch) { throw new Error('Unable to login'); };
```

```
return { user, token: jwt.sign({ userId: user.id }, 'nodesecret') };  
}
```

When it comes to sending back errors about why authentication failed it is best to be as generic as possible which would prevent giving away more information that is really necessary which would make it more difficult for hackers/social engineers from using that information to guess username and passwords to gain access.

If a user logs in with the correct credentials we would return the logged in user data back along with the newly generated authentication token which the client can then use for future requests to the Prisma GraphQL API. We now have an authenticated way to generate tokens when a user logs in or creates a new user account.

The screenshot shows a GraphQL client interface with a top navigation bar containing tabs for various queries and mutations: users, posts, comments, login (selected), createUser, deleteUser, updateUser, createPost, deletePost, and updatePost. Below the navigation bar, there are buttons for 'PRETTIFY', 'HISTORY', and a URL bar showing 'http://localhost:4000/'. A 'COPY CURL' button is also present. The main area is split into two panels. The left panel shows the GraphQL mutation:

```
1 mutation {  
2   login (  
3     data: {  
4       email: "j.doe@email.com"  
5       password: "password123"  
6     }  
7   ) {  
8     user {  
9       id  
10      name  
11    }  
12    token  
13  }  
14 }
```

. The right panel shows the JSON response:

```
{  
  "data": {  
    "login": {  
      "user": {  
        "id": "ck79g6hzc00qa0786m6f413nc",  
        "name": "John Doe"  
      },  
      "token":  
        "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJjazc5ZzZoemMwMHFhMDc4Nm02ZjQxM25jIiwiaWF0IjoxNTgzNTk4NDkxZjQ5X70lKx7lkkR78_sLeCdLi_7lcDjuUdMWXsCd07lYVU"  
    }  
  }  
}
```

. A play button icon is visible between the two panels. On the far right, there are tabs for 'DOCS' and 'SCHEMA'.

Validating Auth Tokens

We now have learned how to create a signup and login mutation which return back an authentication token. We now need to learn how to use the token to authenticate the user i.e. how to pass the token from the Client to the Node.js server and then how Node.js can use this token to perform authentication.

To pass the token from the Client to the Node.js server we would do this via the HTTP Headers. HTTP Headers allow us to pass data from the Client to the server and vice versa regardless of what platform the user uses for example iOS, android, browser, etc. We would use the Authorisation: Bearer structure which is a very common structure for authentication headers.

QUERY VARIABLES HTTP HEADERS (1)

```
1  {  
2    "Authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJjazc5ZzZoemMw"  
3  }
```

This is all we need to do to get the token from the Client to the Node.js server. We need to figure out how we can get access to the JWT from inside of Node.js for the individual mutation resolvers since some mutations would require authentication for example createPost while other mutations would not such as login.

To do this we would make a small change to the context of our server located in the index.js file.

src/index.js:

```
... const server = new GraphQLServer({  
  typeDefs: '...', resolvers: { ... },  
  context (request) { return { db, pubsub, prisma, request }; }  
});
```

The context property can actually also be set to a function and this function then returns the object much like the object we have setup originally. This is an important change which would allow us to access the request headers where the authorisation token actually lives.

This function actually gets called by GraphQL-Yoga with an argument of the request object. This is where we can access the HTTP Headers. The request object argument has a whole bunch of information that make up the request including the raw headers and is the information we need access to. The request.request.headers object property contains all of the headers which is where the authorisation token lives.

Typically speaking there will be whole bunch of methods that would require authentication i.e. for each resolver method we would have to get the header value, parse the authorisation token and then verify the token, etc. We would not want to write out this code for every single Query, Mutation or Subscription that we want to setup for authentication instead we would create a simple utility function that handles all of this for us. We would write the code once and reuse it anywhere we want to setup authentication i.e. DRY principal.

To do this we can setup a new sub-directory in the src directory called utils which would hold all of our utility files. Below is an example:

src/utils/getUserId.js:

```
import jwt from 'jsonwebtoken';

const getUserId = (request) => {
  const header = request.request.headers.authorization;
  if(!header) { throw new Error('Authentication required'); };
  const token = header.replace('Bearer ', '');
  const decoded = jwt.verify(token, 'nodesecret');
  return decoded.userId;
};

export { getUserId as default };
```

Every requests sends back a request and a response headers and this is why we select the request property on the request object to get to the headers and then to the authorisation property.

We can throw an error if there are no header and this would get called whenever the getUserId utility function gets called in the resolvers Query, Mutation or Subscription methods.

There are a tonne of different ways to parse the authentication token to remove “Bearer ” from the header string. One such method is to use the JavaScript .replace string method to replace “Bearer ” with an empty string. Another alternative is the .split string method to split by a space delimiter and we can grab the token value by its index (e.g. `header.split(' ')[1]`).

There are a tonne of different ways to parse the authentication token to remove “Bearer ” from the header string. One such method is to use the JavaScript .replace string method to replace “Bearer ” with an empty string. Another alternative is the .split string method to split by a space delimiter and we can then grab the token value by its index (e.g. `header.split(' ')[1]`).

Now that we have parsed the token from the header we would want to verify it using the `jwt.verify` method to verify that the token is valid for authentication. If the authentication token is valid we would want to return the `userId`.

We can now take advantage of this utility function on any methods we want to use authentication as seen in the below example for `createPost`:

`src/resolvers/Mutation.js`:

```
import getUserId from '../utils/getUserId';

const Mutation = {
  createPost(parent, args, { prisma, request }, info) {
    const userId = getUserId(request);
    return prisma.mutation.createPost({
      data: { ..., author: { connect: { id: userId } } },
      info);
  }
};
```

The mutation method `createPost` in the above example calls on the `utils getUserId` passing the request object along. The `getUserId` method will check if there is a `authorisation` header and if none it will throw an error. If a `authorisation` header exists but is not valid the code will stop there. If the `authorisation` token is valid the `userId` is returned. The mutation will now complete and use the `userId` of the authenticated user as the `author` value when creating the post.

Finally the `schema.graphql` input definition for `CreatePostInput` no longer requires the `author: ID!` property and can be removed completely.

`src/schema.graphql`:

```
type Mutation { createPost(data: CreatePostInput!): Post! }  
type Post { id: ID!, title: String!, body: String!, published: Boolean!, author: User!, ... }  
input CreatePostInput { title: String!, body: String!, published: Boolean! }
```

We now have authentication setup where we can validate the `authorisation` token passed by the Client to the Node.js server via the HTTP Header. We can also use the `authorisation` token to associate the action with the correct `userId`. This utility function can also be used for other resolver methods to add authentication validation easily without having to repeat code.

This completes the basic infrastructure for authentication and we can now lock down the Node.js/Prisma GraphQL API.

Locking Down Individual Type Fields

There are less than obvious ways where someone can get access to data that they should not be able to access. We may lock down individual queries, mutations and subscriptions through authentication while others we may not for example posts.

There is actually a way for users to get access to data without being authenticated by providing a selection set to the queries, mutations or subscriptions which are publicly accessible and has a relationship link to other data. For example a user can get access to draft posts using the public posts query and adding the posts scalar types in the selection type. This is a problem as there are indirect ways of accessing data which generally would have required authentication.

We can close any loopholes in the authentication system and lock down specific type fields in the selection type to avoid the issue above. There are also some edge cases where we would want to lock individual fields such as email addresses being viewable by the outside world.

The issue is not with the queries, mutations or subscription itself but how the operation (selection sets) is resolved. To address this vulnerability we would adjust how some of our fields are actually resolved. Below is an example of the User Type.

The first way to lock down a selection type is to go into the schema.graphql file and adjust the selection type that is returned back for example we can remove the email field from the returned type User to prevent anyone being able to select the email as a return scalar field. However, this is not the approach we would want to take because it may affect other queries which would want to return the

user email when it resolves. So in reality we do not always want to hide the email but we want to hide the email if we are viewing a user other than ourselves the authenticated user.

The alternative approach is to make the email a nullable String scalar type. So when selecting the email sometimes we would get a string if we are logged in as that user and other times we would get null if we are trying to pull the email for every other users. This is the first step to opening the door to not provide the email for certain situations.

src/schema.graphql:

```
type User { id: ID!, name: String!, email: String, ... }
```

We can then go into our custom type JavaScript file e.g. User.js to customise what happens when the Users email field is resolved. So we can set the field as a resolver function and set the code to determine whether or not we should send the email back:

src/resolver/User.js:

```
import getUserId from '../utils/getUserId';  
const User = {  
  email(parent, args, { request }, info) {  
    const userId = getUserId(request, false);  
    if(userId && userId === parent.id) {  
      return parent.email;  
    }  
  }  
}
```

```
        } else { return null };  
    }  
};  
export { User as default };
```

In the resolver function we can now send back either null or a String scalar type without running into any errors. To get access to the information we have access to the parent, so in the above case the parent is the User's object. We can therefore, check if the user is authenticated and then check if the authenticated id matches with the parent.id property. If this does match then we know a user is trying to select their own email address in which case we can return. If it does not match up then we know that the user is trying to select a different user's email and then we can return null.

We now have a way to individually hide fields based off of whether or not a user is authenticated as some user. This is a great approach; however, there is a serious flaw with the current code which may not be noticeable until you run into the edge case.

What actually happens when we fire off the users query? The first thing that happens is that it heads off over to the Query.js file and the users method runs i.e. the one associated with the method. The users query method really returns for every user whatever information was selected — in the below screenshot we decided to select the id and email field and are no longer receiving the name back because it is not selected.

If we are logged in as a user we should be able to see the authorised user's email. Data for one field should not change because another field is being selected or unselected and this is a problem. This will be addressed in the next section on GraphQL Fragments.

Fragments

A fragment allows us to create a reusable selection set so that we can define what we want once and we can use it anywhere where we want those selections. To create a fragment we use the fragment keyword followed by the name of the fragment (similar to a variable name) and then the on keyword. For each fragment it must be associated with a specific type. The selection types we want selecting on the associated type is selected within the opening and closing curly brackets. Below is an example syntax for the User Type:

```
fragments userFields on User { id name email }
```

The above is a valid fragment because the selection type fields exists on the User Type. If we selected a field that did not exist on the Type this will throw an error because we are specific on the Type we are selecting from and can validate the operation before even performing it.

To use a fragment, anywhere we provide a selection set, we can use the following syntax of ... followed by the fragment name. Below is a screenshot example of using fragments within the GraphQL Playground:

The screenshot shows the GraphQL Playground interface. The top bar contains tabs for various queries: 'users', 'me', 'posts', 'post', 'myPosts', 'comments', 'login', 'createUser', 'deleteUser', and 'updateUser'. Below the tabs are buttons for 'PRETTIFY', 'HISTORY', and a URL bar showing 'http://localhost:4000/'. On the right side, there are buttons for 'COPY CURL', 'DOCS', and 'SCHEMA'.

The query editor on the left contains the following GraphQL query:

```
1 query {  
2   users {  
3     ...userFields  
4   }  
5 }  
6  
7 fragment userFields on User {  
8   id  
9   name  
10  email  
11 }
```

The response editor on the right shows the JSON response:

```
{  
  "data": {  
    "users": [  
      {  
        "id": "ck7i1064100100886a30ixuj0",  
        "name": "John Doe",  
        "email": "j.doe@email.com"  
      },  
      {  
        "id": "ck7i1c3wn001w0886f65lwmno",  
        "name": "Betty Doe",  
        "email": null  
      }  
    ]  
  }  
}
```

In essence we are creating a fragment variable that we can use wherever we want to grab the selection type fields off of a User Type. If we were to change the fragment to select only the id and name we would get back just the id and name from the fragment and so on.

Alongside of our fragments we can select other fields explicitly and can therefore use a combination of the two i.e. explicitly selecting selection types or using fragments to select selection types.

In the fragments we can also choose to select related data such as the posts type fields.

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

```
1 query {  
2   users {  
3     ...userFields  
4     email  
5   }  
6 }  
7  
8 fragment userFields on User {  
9   id  
10  name  
11  posts {  
12    id  
13  }  
14 }
```

The response editor on the right shows the JSON response:

```
{  
  "data": {  
    "users": [  
      {  
        "id": "ck7i1064100100886a30ixuj0",  
        "name": "John Doe",  
        "posts": [  
          {  
            "id": "ck7i10zfu00170886g0tybtb6"  
          }  
        ],  
        "email": "j.doe@email.com"  
      },  
      {  
        "id": "ck7i1c3wn001w0886f65lwmno",  
        "name": "Betty Doe",  
        "posts": [  
          {  
            "id": "ck7i1wc52002y0886p4q83xb5"  
          }  
        ],  
        "email": null  
      }  
    ]  
  }  
}
```

Using fragments will allow us to avoid re-typing the same selection set over and over again. Inside of GraphQL Playground there is no way to share the fragments between the various queries and mutations but that is OK because we are still able to use fragments inside of the Prisma application to solve the problem we identified in the last section.

Going back to the User.js file we can integrate fragments into our application. When we provide the resolvers for fields like the email we can also choose to provide a fragment which will make sure that Prisma fetches certain data about, in this scenario the User, when a specific resolver runs. This will force Prisma to get the id from the database even if the Client did not request it. Below is an example of implementing this:

src/resolver/User.js:

```
import getUserId from '../utils/getUserId';

const User = {
  email: {
    fragment: 'fragment userId on User { id }',
    resolve (parent, args, { request }, info) {
      const userId = getUserId(request, false);
      if(userId && userId === parent.id) {
        return parent.email;
      } else { return null };
    }
  }
}
```

```
};  
export { User as default };
```

Email is no longer a function but an object which has two properties the first is the fragment which has a string value of the fragment syntax seen above and the second is the resolve which is a function i.e. the same function we had before.

We do not need to select the email in the fragment because the User resolver will only run if the Client asked for the email and therefore we would not need to ask for it.

With this in place we need to do a little refactoring. While Prisma does support fragments in this way it requires a little configuration changes to how we setup the server in order to add support for fragments which is essential for locking down data. We would need to focus on the prisma.js and index.js files.

At the end of the day the goal is to grab a single function from prisma-binding called extract fragment replacements which is a function that gets called with our resolvers. What it gives us back is something that we would pass into our prisma constructor function in prisma.js and our GraphQLServer constructor function in index.js. This is what enables the syntax we would use as seen in the User.js file.

To setup the architecture we would refactor our index.js server's resolvers into a separate file for where

our resolvers get defined. We can then load that in where necessary. To start this refactoring we would create a new file in our resolvers directory called index.js. This file is in charge of getting the resolvers object built.

src/resolver/index.js:

```
import { extractFragmentReplacements } from 'prisma-binding';
import Query from './Query';
import Mutation from './Mutation';
import Subscription from './Subscription';
import User from './User';
import Post from './Post';
import Comment from './Comment';
const resolvers = { Query, Mutation, Subscription, User, Post, Comment };
const fragmentReplacements = extractFragmentReplacements(resolvers);
export { resolvers, fragmentReplacements };
```

The resolvers object is constructed in this file and we can import it to the src/index.js file and assign it to the resolvers property (resolvers: resolvers) using the ES6 shorthand syntax as seen below:

src/index.js:

```
import { resolvers } from './resolvers/index';
const server = new GraphQLServer({ typeDefs: '...', resolvers, ... });
```

At this point we have not changed the functionality of the application at all and will continue not to work as we have not setup support for fragments but we now have an application architecture that will actually allow us to setup support without running into circular referencing between the src/index.js and src/prisma.js files.

Inside of the src/resolvers/index.js file we would import the extractFragmentReplacements function from prisma-binding. We can now call on this function and pass in the resolvers and since the resolvers are defined in this file it is going to be easy to implement. We would create a new variable and call on the extractFragmentReplacements function passing in the resolvers object. We can also export the new variable from this file and import it into both the src/prisma.js and src/index.js file and pass them into the necessary constructor functions.

All of this refactoring is to make this possible. The new variable fragmentReplacements is nothing more than a list of all of the GraphQL fragment definitions (in this example we only have one defined in User.js). This allows to specify the fields that are required for the resolver function to run correctly i.e. the email resolver function requires the id from the User Type to run correctly.

src/prisma.js:

```
... import { fragmentReplacements } from './resolvers/index';  
const prisma = new Prisma({ typeDefs: '...', endpoints: '...', secret: '...',  
  fragmentReplacements });
```

We can add the fragmentReplacements import to the property called fragmentReplacements i.e. we

can use the ES6 shorthand syntax. We can do exactly for the same for the src/index.js file:

src/index.js:

```
... import { resolvers, fragmentReplacements } from './resolvers/index';  
const server = new GraphQLServer({ typeDefs: '...', resolvers, context(request): '...',  
    fragmentReplacements });  
server.start(( ) => { console.log('The server is up on http://localhost:40000' )});
```

Both the Node.js server and Prisma are configured to actually use the fragment we setup in User.js and can test things out to see that they are working correctly in the GraphQL Playground.



We are now able to pull out the email address of the authenticated user only which solves the issue we previously had when relying on the Client to provide the selection type. We are now able to create custom resolvers to do something more meaningful and close any unwanted loopholes securing our data through authentication.

Token Expiration

When setting up a JSON Web Token we can set it up to be valid for a certain period of time. The jsonwebtoken library has built in support for token expiration. The only thing we need to do is make a small change to how we call the .sign method. This method has a third optional argument we can provide which is an options object and in here we can specify how long the token should be considered valid.

The expiresIn options property lets us setup a specific amount of time in which the token should expire and we express it in a human readable format for example '2 days', '2d', '10h', etc. If we specify a number without the unit this will default to milliseconds. We can see all of the units available to us by visiting the library that the jsonwebtoken library uses as a dependency called ms on <https://github.com/zeit/ms>. Below is an example:

src/resolvers/Mutation.js:

```
... const Mutation = {  
  async createUser(parent, args, { prisma }, info) { ...  
    return { user, token: jwt.sign({ userId: user.id }, 'nodesecret', { expiresIn: '1 day' }) } }  
};
```

When an expired token is used with authentication this will throw an error of 'jwt expired' preventing it from being used any further. With this built in feature all we have to do is configure the expiresIn option when creating the token and do not need to make any changes to our application code when verifying it because the .verify method is going to verify everything for us.

Password Updates

We can now finally look at updating our mutation to allow users to update their password to round off the topic on Authorisation.

The first thing we need to do is update the argument list for updateUser mutation type accepts in the schema.graphql file. We would add password as an optional input:

src/schema.graphql:

```
type Mutation { ..., updateUser(date: UpdatUserInput!): User! }  
input UpdateUserInput { name: String, email: String, password: String }
```

We can create a utility function in the utils directory which would be responsible for validating and hashing passwords, implementing the principal of DRY (Don't Repeat Yourself).

src/utils/hashPassword.js:

```
import bcrypt from 'bcryptjs';  
const hashpassword = (password) => {  
  if(password.length < 8) { throw new error('Password must be 8 characters or longer'); }  
  return bcrypt.hash(password, 10);  
};  
export { hashpassword as default };
```

We now have a reusable hash password function which would still validate the length of the password and also still actually returns the hashed password value. We can use this function in the Mutation.js file for wherever we need it i.e. creating a user or updating a user's password.

src/resolvers/Mutation.js:

```
import hashPassword from '../utils/hashPassword';

const Mutation = {
  async createUser(parent, args, { prisma }, info) {
    const password = await hashPassword(args.data.password);
    const user = prisma.mutation.createUser({ data: { ...args.data, password } });
    return { user, token: generateToken(user.id) };
  },
  async updateUser(parent, args, { prisma, request }, info) {
    const userId = getUserId(request);
    if(typeof args.data.password === 'string') {
      args.data.password = hashPassword(args.data.password);
    };
    return prisma.mutation.updateUser({ where: { id: userId }, data: args.data }, info);
  }
};
```

The mutations are doing exactly what they did before in terms of functionality but we have refactored it to be able to use the validation and hashing functionality.

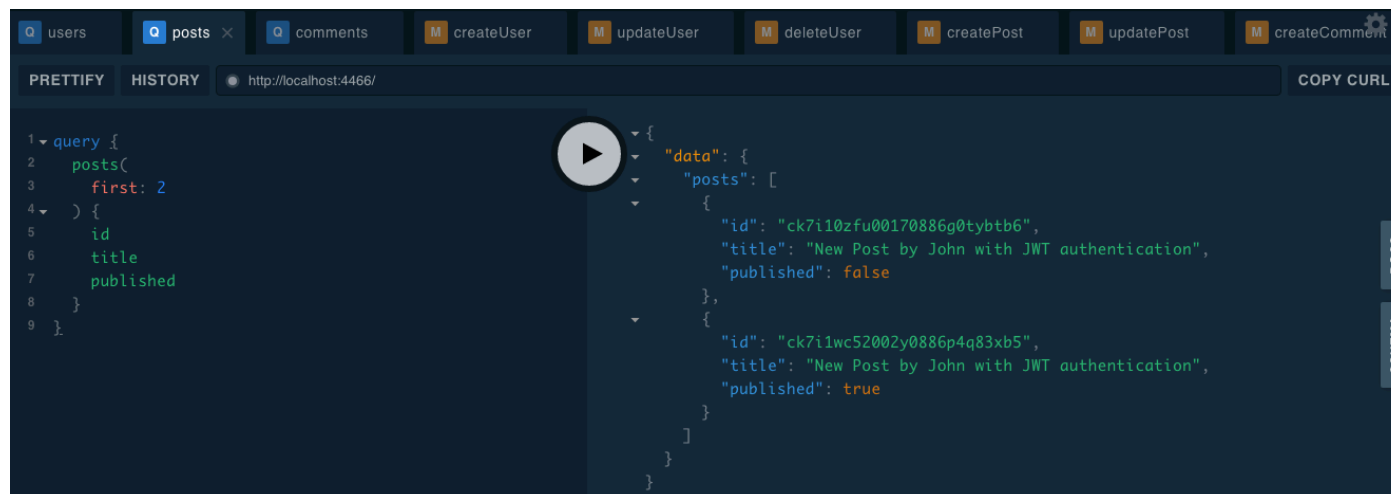
For the updateUser mutation we are checking whether a password was provided by checking its type which should be a string if one was provided. This password is provided to the hashPassword utility function which overrides the plain text password provided in the argument and then gets saved to the database when the updateUser mutation resolves updating the user's data. This now provides a system to update the user's password.

We now have a reusable way to validate and hash passwords allowing us to create and update users through the existing mutations. This wraps up the topic of setting up an authentication system architecture with GraphQL protecting our database data from the outside world and only allowing authenticated users to perform specific read/write operations to the database.

Section 6: Pagination and Sorting with GraphQL

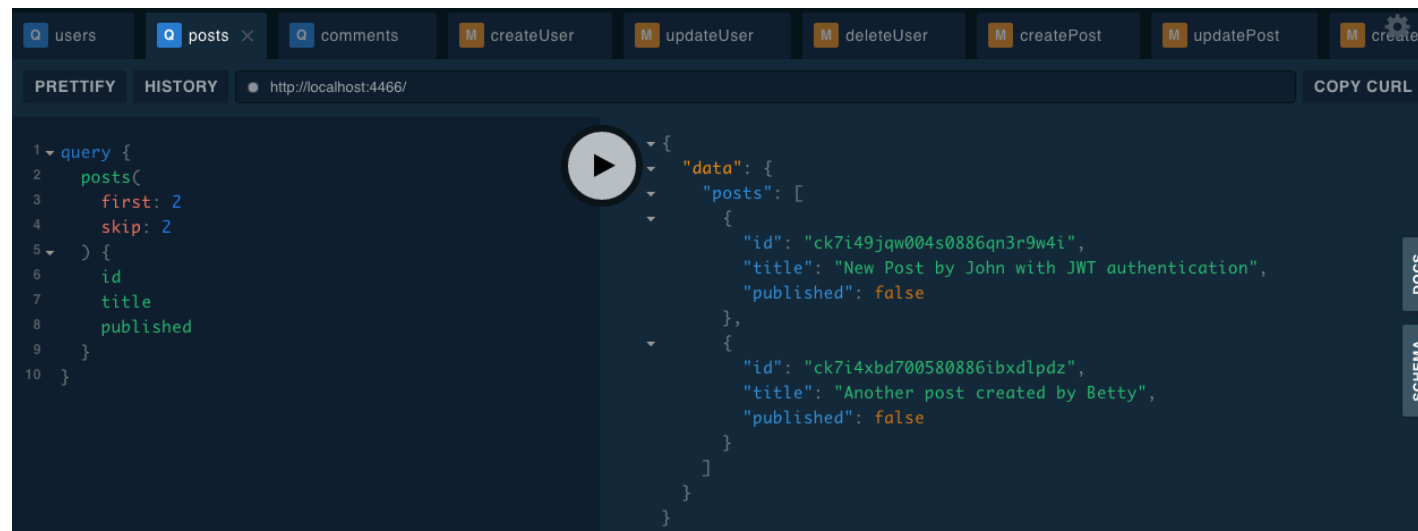
Pagination

Pagination provides Clients the ability to fetch a subset of the data in the database for example allowing a user to fetch the first 10 posts from a database containing 100+ posts. The user can make requests for more when they are ready which can either be done automatically or manually via buttons. Pagination is a very important feature of any application that is expected to scale.



Prisma provides us with 5 arguments for our queries which are skip, after, before, first and last. The first argument takes in an integer value and allows us to return the first x number of data where x is the integer value we pass in. By default Prisma limits us to a 1000 integer value.

Using the first query argument allows us to limit how many records are being seen. This alone is not particularly useful because if we want to load the next 2 records we would need to use a second argument called skip which takes in an integer value. Skip tells Prisma how many records to skip and



The first argument will pickup the next first records after the skip. Therefore, using a combination of first and skip allows us to hone in on exactly the data we want to access. This allows us to setup basic pagination.

To set this up in our Node.js server we would need to add code to our `schema.graphql` and `Query.js` files. Any Query that ends up sending back an array should support pagination. We would add the first and skip arguments to our Query Types:

[src/schema.graphql:](#)

```
type Query { users(query: String!, first: Int, skip: Int): [User!]! }
```

The first and skip are optional integers and at the end of the day we will pass these arguments through to Prisma and Node.js is not going to do anything meaningful with them at all. We can go into `Query.js` and make use of the arguments:

[src/resolvers/Query.js:](#)

```
const Query = { users(parent, args, { prisma }, info) {  
  const opArgs = { first: args.first, skip: args.skip }
```

```
}}
```

In the opArgs argument object we would setup the first and skip arguments and values. We should now have pagination all setup. We can now use pagination with the users query on our API located on <http://localhost:4000>.