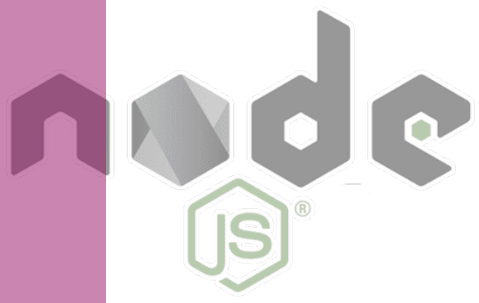
GraphQL

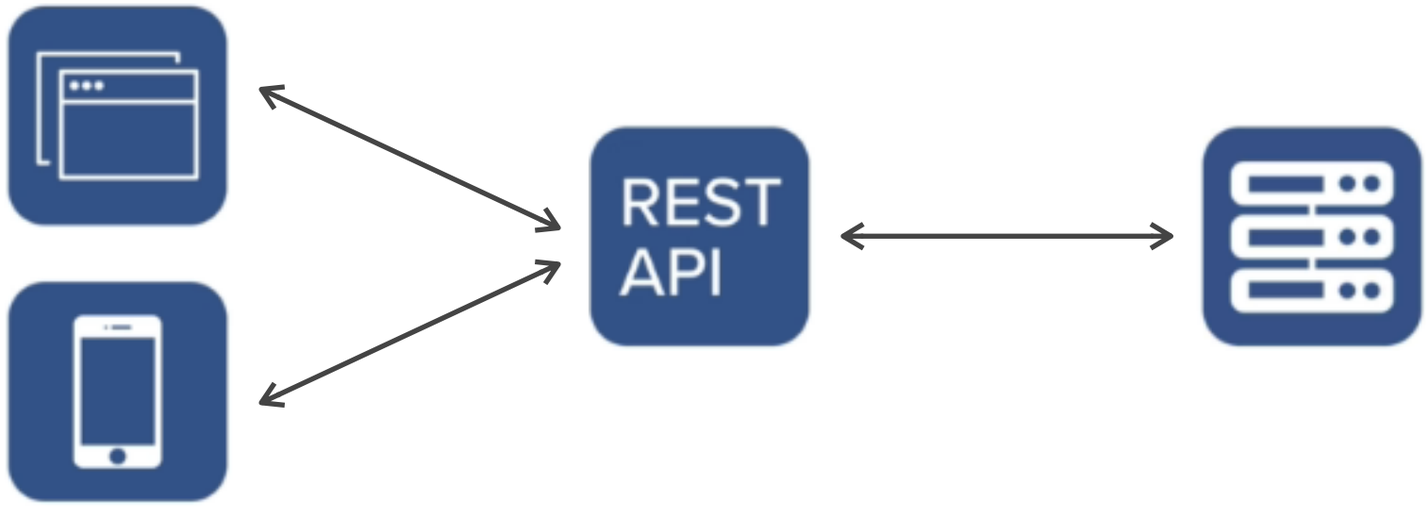


GraphQL with Node.js and Apollo

Section 1: Introduction

Why GraphQL & Installing Node.js and Visual Studio Code

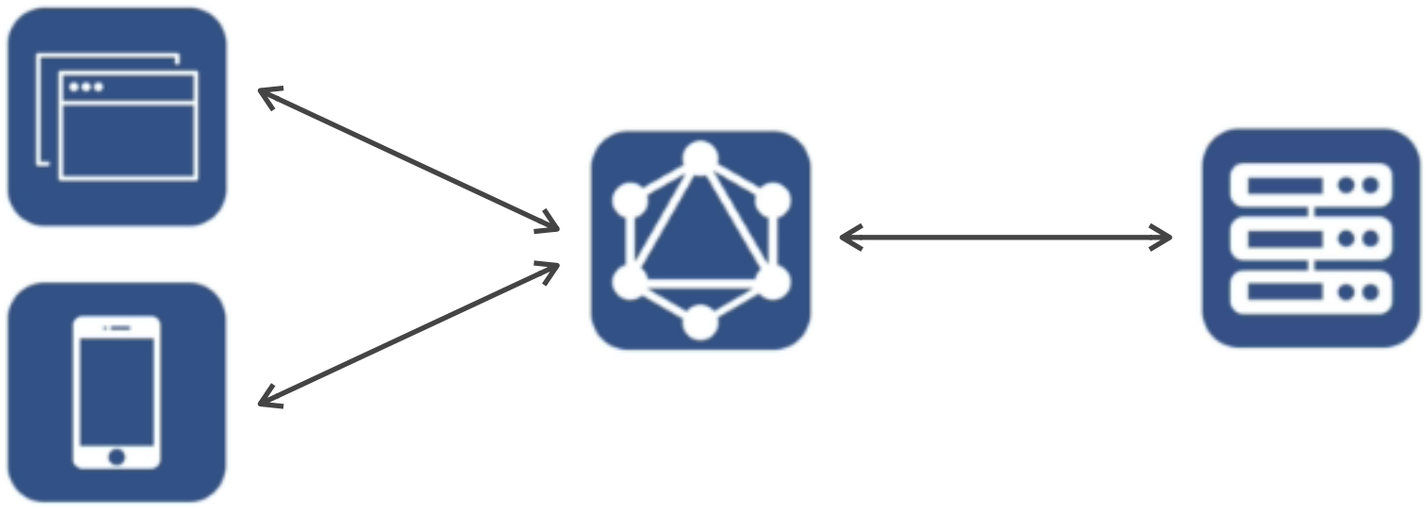
Most current applications likely uses some form of a REST API where we are making HTTP requests between the client and the server to send and receive data. Below is a visual representation of a REST API application where we have a client and a backend database/server.



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.

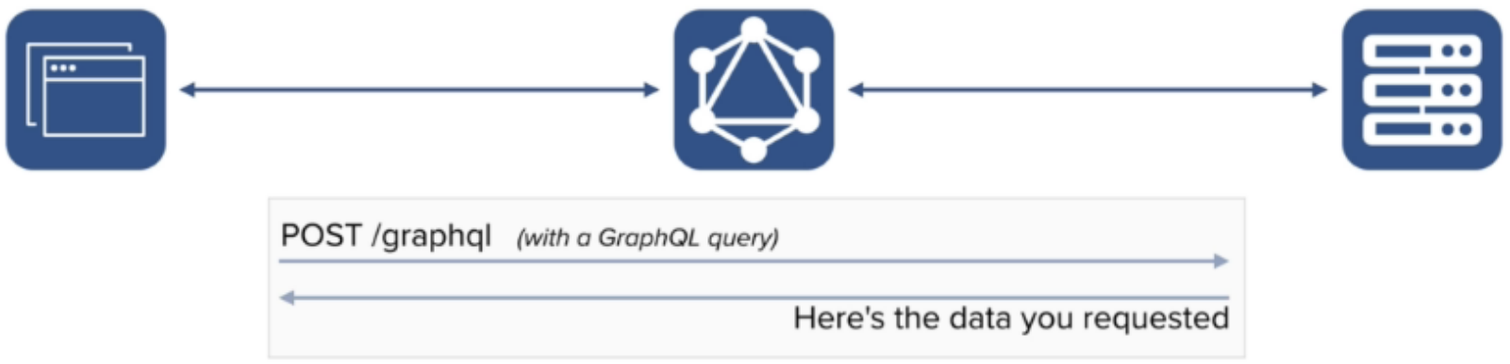


Why use GraphQL? GraphQL is fast, flexible, easy to use and simple to maintain. Below is a visual representation of a typical example of loading the necessary data for a page with a REST API and then with a GraphQL API.

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 bog 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.

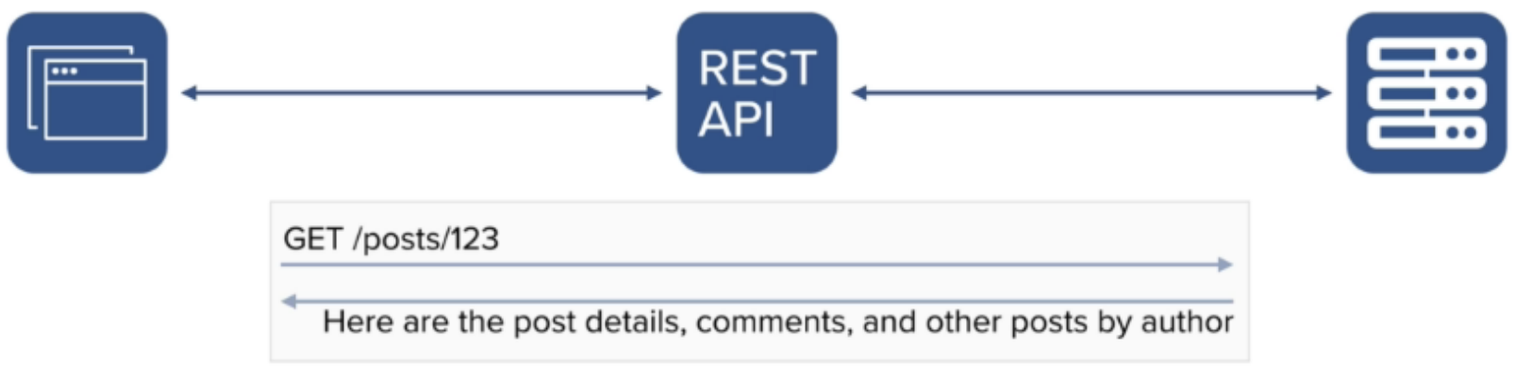
The GraphQL API exposes a single endpoint which is a very important piece to the puzzle. This endpoint could be called anything we like (in the example below it is named /graphql). When making a HTTP request to this single endpoint we would also send along a GraphQL query.



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.

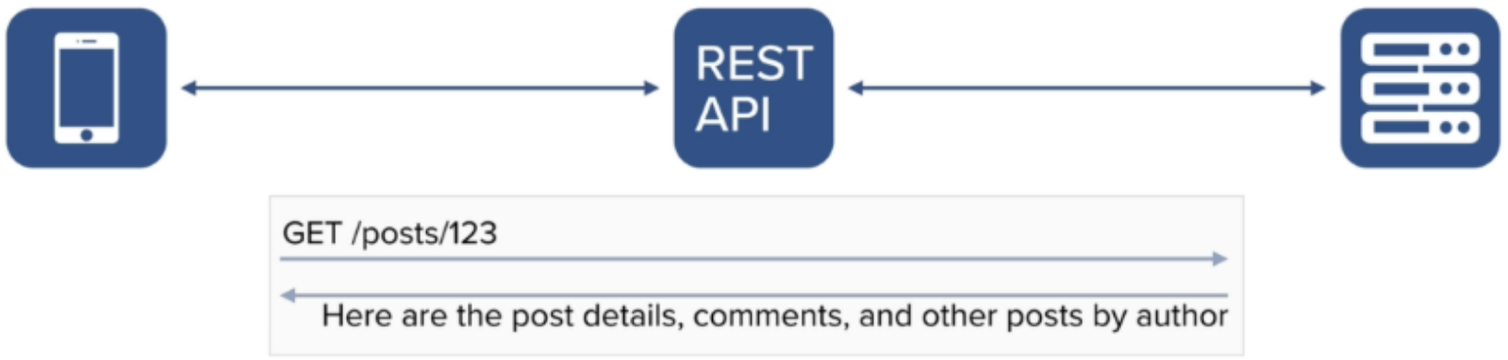
The important difference between a REST API and a GraphQL API is that the latter allows the client to determine what data it gets back as apposed to the former HTTP endpoint which allows the server to determine what data comes back from and endpoint. Clearly 3 HTTP requests is more than a single HTTP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

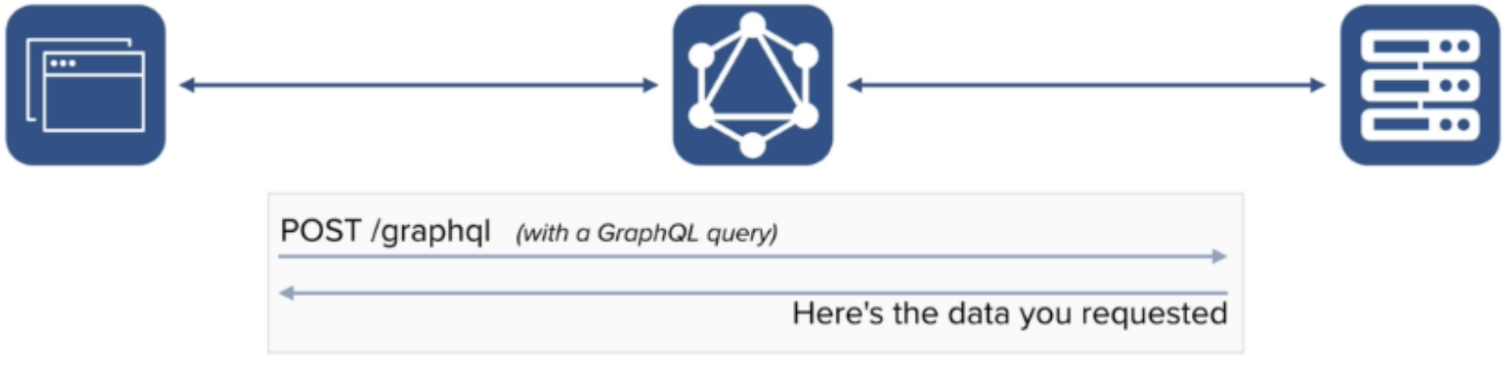


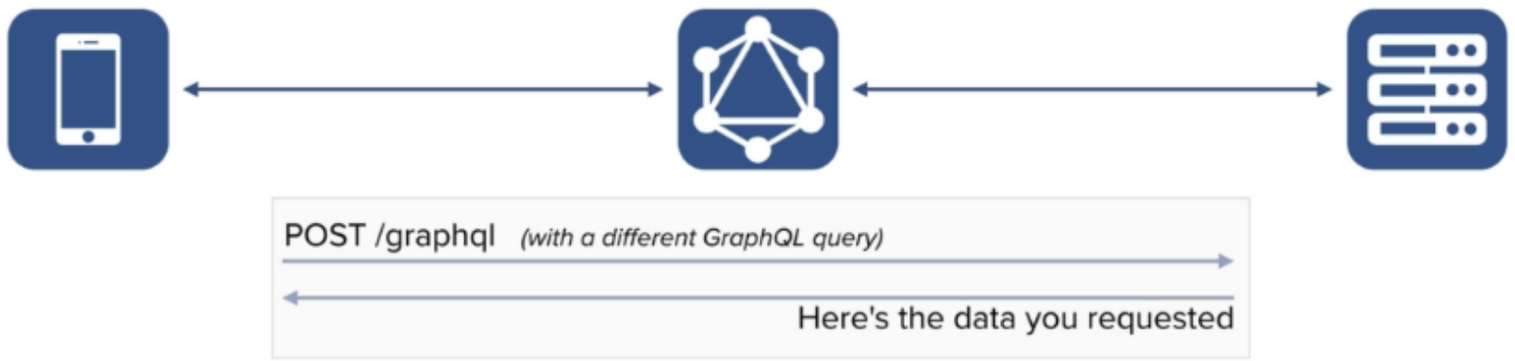
The above would be a perfectly fine approach and it would give the client everything it needs to render that page with just a single HTTP request. However, the problem with this solution is that we now have this one endpoint which is making way more database request than it was before.

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 queery 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:

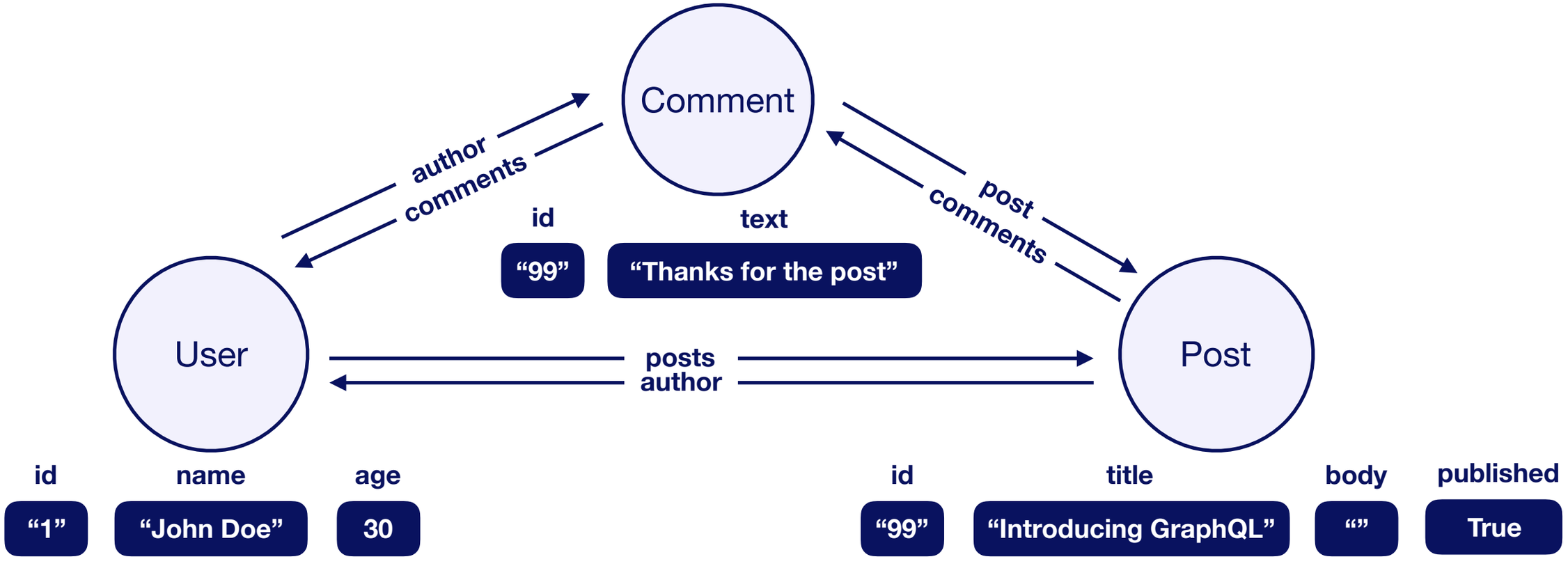
Babel ES6/7 (dzannotti), Beautify (HookyQR), Docker (Microsoft), Duplicate action (mrminc), GraphQL for VSCode (Kumar Harsh), npm (egamma), and npm Intellisense (Christian Kohler).

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 filed 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 if 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 a 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 cmder at https://cmder.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.