Get Started With...

GraphQL

Learn how to build a simple GraphQL API.

Donovan Hutchinson

GetStartedWith.dev

Contents

What is GraphQL

GraphQL is a query language that is designed to be a better way of requesting data from APIs.

In recent years, most web-based APIs worked in a style called REST. With this approach we would interact with servers to request and send data using different URLs that defined the kind of response we wanted.

With REST we would make GET requests for data, PUT or POST requests to send data to the server to make changes. We'd use different URLs for different sets of data.

GraphQL is different. It still allows us to make requests to our API server, but it allows us to more carefully describe the shape of the data we want back. In doing so, we can ask for just the data we need, and reduce the number of requests needed to request complex data sets.

GraphQL vs REST

For example, with REST we could send a GET request to a `movies` API to obtain a list of newly released movies. We would send a `GET` request to something like `oursite.com/api/movies/new` and it would give us a list of movie IDs.

If we wanted specific movie details on each of those IDs, we would need to then assemble at least one more query to ask for each of the movies by ID.

This would often mean our front-end spends a lot of time sending out multiple requests, and then the data we get back might be larger than we need, wasting bandwidth.

With GraphQL we could cut back on the number of requests.

If our API had a list of movies, and a list of actors that cross-referenced the movies list, we can request only the data we need by sending a request like this:

```
{
  movies {
    title
    starring {
      name
    }
  }
}
```

This request returns the data we are looking for in the form of an array of movie objects containing `title`, and a list of names of the actors starring in the movies.

With REST we would have had to make several requests, one to the `movies` database, and perhaps several to the `actors` database.

By writing relational queries like this, we can achieve more in a single request, save on bandwidth and write less code.

In this guide I'll mostly be focusing on how we can make use of GraphQL with JavaScript and web apps, but GraphQL is not limited to any one technology. GraphQL is a specification, created by Facebook, that defines ways to access data. This means how we use it depends on what we want to build.

Implementations of the GraphQL API are available from many sources and we have many tools to choose from including server-side and client-side implementations built with most languages, not just JavaScript.

You can explore some of these options on the GraphQL code section.

When to use GraphQL

GraphQL is useful when you are building web apps that load data from a back-end API. If you have control of the back-end, you can make use of GraphQL to define ways your front-end can request data.

This can make building your web applications faster to develop and result in lighter, faster queries.

Sometimes we may have to use an external REST-based API. In these situations it might be worthwhile building an in-between GraphQL layer to allow our front-end code to make use of GraphQL queries, which are then handled with the necessary REST requests by the back-end.

Let's see how we can build our own GraphQL server, explore some ways of querying a database, and then see how this can translate into a simple web application.

Build a GraphQL server

Let's make a simple server that we can then make queries on and explore the different approaches GraphQL gives us.

A GraphQL server is made up of two main parts.

Schema

A GraphQL schema is how we describe the shape of our API. It is made up of a list of type definitions that describe what queries we allow, and the types of objects we are working with.

A schema can look like a list of definitions like this:

```
type SomeObject {
  id: String!
  name: String!
}

type Query {
  getObject(id: String!): SomeObject
}
```

Resolvers

With a schema in place, we need to build the actual response. To do this we use `resolvers`. These take the form of functions that return the necessary data for our queries. In the above schema, we might have a resolver like this:

```
const resolvers = {
  getObject: (id) => getObjectFromDatabase(id);
}
```

This describes the function that will be called when the `getObject` query is received.

We will explore how to build a schema and the asociated resolvers more when we set up our API.

That's enough of the theory for now. Let's learn how to set up a real schema with resolvers and explore our own GraphQL API.

Set up Node.js, GraphQL and Apollo

If we haven't already done so, you should install Node.js. This will give us `npm` commands as well as letting us run our JavaScript code locally.

With that done, make a new directory for our project, navigate to that directory in your command line, and then run:

```
npm init --yes
```

This will initialise our project and create a `package.json` file that helps define the project and any dependencies. Hit return a few times for the default values.

Then install our dependencies:

```
npm install apollo-server graphql -S
```

This installs a few things we need.

This will install apollo-server, which will handle serving our GraphQL API, along with GraphQL, a JavaScript-based implementation of the GraphQL query language.

Create our Node.js server

Creating our own Node.js server is as simple as creating a JavaScript file, `index.js` and adding the following code:

```
const { ApolloServer, gql } = require('apollo-server');
const db = require('./database.js');

// Construct our request types, using GraphQL schema language
const typeDefs = gql`
  type Query {
    hello: String
  }
`;

const resolvers = {
  Query: {
    hello: () => {
      return 'Hello world!';
    }
  }
};

// The ApolloServer constructor requires two parameters: your
schema
// definition and your set of resolvers.
const server = new ApolloServer({ typeDefs, resolvers });
```

```
// The `listen` method launches a web server.
server.listen().then(({ url }) => {
  console.log(`🚀  Server ready at ${url}`);
});
```

Let's walk through this code and explore what it does, before we see it in action.

First we use `require` to call in the modules we need for this server to work. These are the same modules we installed earlier.

Then we define two important GraphQL components, the `typeDefs` and `resolvers`.

Schema and resolvers

In our `typeDefs` variable we set up a string that describes the schema for our GraphQL API.

A schema is like a description of the "shape" of the API. It defines what requests we can make and what type of response to expect.

To begin with we have a definition for `Query`. This is a container into which we put the type definition of any queries we want to make.

The first query we've designer is called `hello`, and it should give us a response of type `string`.

We'll adjust this schema later to give allow us to test more of the GraphQL functions.

Next we set up a resolver function. This is an object containing functions that map to the queries defined earlier. In this case, we set up a similar `Query` container and in it, a method called `hello`.

This `hello` function is a simple one that returns a string (`Hello world!`). In a real implementation this could call out to our API and query a database, before returning the expected data.

Lastly we define our `ApolloServer` app. By default this will run at the root of `localhost`, on the default port, `4000`.

Running our GraphQL server

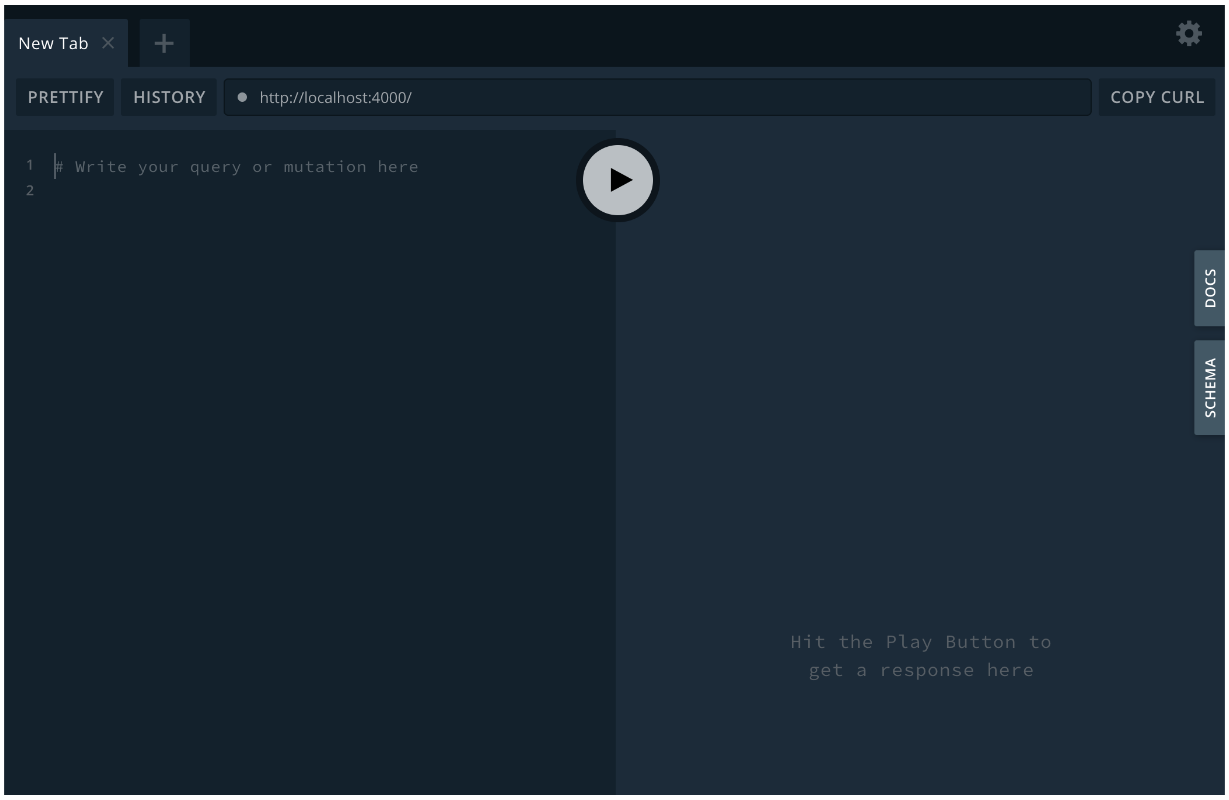
Now that we've set up a simple server, let's run it.

```
node index.js
```

You should see some output in the command line like:

```
🚀  Server ready at http://localhost:4000/
```

If you open your web browser and go to the address `http://localhost:4000/` you should see the Apollo web interface.



```
# Write your query or mutation here
```

Hit the Play Button to get a response here

Hello GraphQL world

Let's try out our new GraphQL server by sending it a query.

On the left panel we find some comments (starting with `#`). Replace the comments with the following query:

```
{
  hello
}
```

Then press the Play (▶) button (or use `Ctrl + Enter`) to run our query. This should activate our `hello` resolver function, which generates the following response:

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

Here we can see that when we sent the query containing the object with `hello`, we received back a JSON object that has a similar shape but with the result of our query added.

GraphQL queries

Now that we have a server running, we can explore some more GraphQL queries. Make sure to bookmark the official GraphQL docs so that you can check how things work as well as find out about new features.

At the movies

To better explore how GraphQL queries work, we're going to set up a schema along with some new resolvers and give them some data to work with.

Note: If you prefer the finished code, you can find this finished example on GitHub.

In our index.js file begin by replacing the `const typeDefs...` section with the following:

```
const typeDefs = gql(`
  type Actor {
    id: String!
    name: String!
    appearsIn: [Movie!]!
  }

  type Movie {
    id: String!
    title: String!
    starring: [Actor!]!
  }

  type Query {
    actors: [Actor]
    movies: [Movie]
    actor(id: String!): Actor
    movie(id: String!): Movie
  }
`);
```

We have introduced two new `type` definitions here, `Actor` and `Movie`. By convention we start these with capital letters so that it's easier to tell that we're working with custom object types.

The `Actor` definition states that each Actor object should have an `id` along with a `name`, which are strings, along with an array of `Movie` objects which we call `appearsIn`.

The exclamation points you see in the definition are a way of saying these are required. Each `Actor` object requires a string for `id` and `name`, and should appear in at least one `Movie`.

Similarly, we give the `Movie` object it's own definition.

Then we expand our list of `Query` definitions. In GraphQL, we can have two main types of queries, referred to as `query`, `mutation` or `subscription`. We're starting with the first type but will explore the others later.

To give us the ability to query our new data, we've added `actors`, `movies`, `actor` and `movie` queries, replacing the `hello` query we had before.

These new queries will allow us to retrieve an array of actors, movies, or a single one of each.

Adding data and resolvers

With these new queries added, we need to build `resolvers` to connect our data to the queries.

But wait, we need data! If you like you could hook up a database here and use your resolvers to access the database as needed. However for simplicity I'm going to set up a fake database as a static file.

To set it up download the content of this gist and save it as `database.js`.

The `database.js` file exports an object that looks a bit like this:

```
{
  actors: {
    a01: {
      id: 'a01',
      name: 'Harvey Keitel',
      appearsIn: [
        movies.m01
      ]
    }
  },
  movies: {
    {
      id: 'm01',
      title: 'Reservoir Dogs',
      starring: [
        actors.a01
      ]
    }
  }
}
```

We have two sets of data, containing `actors` and `movies`. Each references the other, which we will use to our advantage when creating queries.

Back in your `index.js` file, we need to bring in our database. Add this line near the top of the file:

```
const db = require('./database.js');
```

We can now update our resolvers, replacing them as follows:

```
const resolvers = {
  Query: {
    actors: () => Object.values(db.actors),
    movies: () => Object.values(db.movies),
    actor: (parent, { id }) => db.actors[id],
    movie: (parent, { id }) => db.movies[id]
  }
};
```

These new resolvers follow the same pattern as the `Query` type we defined earlier. With these in place we should be able to try some GraphQL queries.

Simple query

In our browser, running at `http://localhost:4000/`, we should see our Apollo Playground.

On the left side, replace the content with this query:

```
{
  movies {
    title
  }
}
```

This is a simple GraphQL query. It tells our API that it would like to call the query `movies`, and the shape of the data we expect back is `{ title }`.

Pressing Play (▶) button and we should see the following JSON result on the right side:

```
{
  "data": {
    "movies": [
      {
        "title": "Reservoir Dogs"
      },
      {
        "title": "True Romance"
      },
      {
        "title": "Pulp Fiction"
      }
    ]
  }
}
```

In the result we receive a `data` object that lists out our query and the data returned by our query.

What if we wanted more than just titles? We can adjust our query:

```
{
  movies {
    id
    title
  }
}
```

This should give us a list that also includes the `id` field from each of our movies. GraphQL can automatically return any data if the fields match the properties.

In this case, our `movies` objects contain `id` and `title`, so GraphQL is smart enough to know that we want that data from each object.

Complex queries

As we shape our queries we can create more complex shapes. Let's add the `starring` field.

```
{
  movies {
    id,
    title,
    starring {
      name
    }
  }
}
```

Since `starring` is an array of `Actor` objects, GraphQL will automatically return each of these objects to us. We can then specify what we want from each of these objects. We know from our definition that each `Actor` has a `name` and we can request it.

The result should now include a list of actors!

```
{
  "data": {
    "movies": [
      {
        "id": "m01",
        "title": "Reservoir Dogs",
        "starring": [
          {
            "name": "Harvey Keitel"
          },
          {
            "name": "Tim Roth"
          },
          {
            "name": "Steve Buscemi"
          },
          {
            "name": "Quentin Tarantino"
          }
        ]
      },
      ...
```

Try playing around with the queries. You can change `movies` to `actors` and explore the data from that angle too.

Arguments

So far we have created wide queries that return all our movies and actors. What if we only wanted one specific movie? Let's use our `movie` query along with an `argument` to specify just one result.

We can pass an argument by adding `(id: "value")` to our query like so:

```
{
  movie(id: "m01") {
    title
  }
}
```

In the above we specify the movie with `id` of `m01`. It should give us this result:

```
{
  "data": {
    "movie": {
      "title": "Reservoir Dogs"
    }
  }
}
```

But what if we wanted more than one movie? We might try something like this:

```
{
  movie(id: "m01") {
    title
  }
  movie(id:"m02") { # ERROR
    title
  }
}
```

However this won't work. The two identical `movie` fields are in conflict. To fix this we can use `aliases`.

Aliases

To query more than one movie at a time we need to give each query a unique name, or `alias`. Let's adjust the above query to call the first movie `movie1` and the second, `movie2`:

```
{
  movie1: movie(id: "m01") {
    title
  }
  movie2: movie(id:"m02") {
    title
  }
}
```

This should give us a result:

```
{
  "data": {
    "movie1": {
      "title": "Reservoir Dogs"
    },
    "movie2": {
      "title": "True Romance"
    }
  }
}
```

This works nicely and might be fine for simple queries, but if we were to request more than just `title` from each movie, we'd end up with a lot of repetition. For example:

```
{
  movie1: movie(id: "m01") {
    title
    starring {
      name
    }
  }
  movie2: movie(id:"m02") {
    title
    starring {
      name
    }
  }
}
```

The above works but can quickly become difficult to manage. We can simplify this query using `fragments`.

Fragments

A fragment is a reusable set of fields that we can apply to our queries. It takes the form of `fragment foo on Object`, where `foo` is whatever name we want to give it and `Object` is one of the types of objects we have defined such as `Movie` or `Actor`.

We can improve our alias example above by using fragments:

```
{
  movie1: movie(id: "m01") {
    ...movieFields
  }
  movie2: movie(id:"m02") {
    ...movieFields
  }
}

fragment movieFields on Movie {
  title
  starring {
    name
  }
}
```

In this query we are using spread syntax (...) to expand the fields from the `movieFields` fragment.

Since we only define the `movieFields` fields once, it removes some of the repetition in our query.

Operation name

Each of our queries so far have been using a shorthand. When we are working with more queries, we can be more specific and use the full `operation type` as well as giving each query an `operation name`.

```
query AllMovies {
  movies {
    title
  }
}

query AllActors {
  actors {
    name
  }
}
```

Using names operations is recommended in production code as it can help when debugging as error messages can point to specific named queries.

If we supply our browser with the above queries, we can run them one at a time as the "Execute Query" button will give a dropdown of each query.

Variables

When we use GraphQL in production we won't want every query to perform one specific task. We might want our `movie` query to be able to query any of our movies, rather than just one specific one.

This means avoiding hard-coding the movie ID. Let's make a dynammic, reusable query using `variables`.

First we set up our query:

```
query MovieQuery($movieId: String!) {
  movie(id: $movieId) {
    title
  }
}
```

This query now has a new addition, the `($movieId: String!)` portion added to the operation name. This tells the query to expect a `movieId`, and that it should be a required string.

Next, our query makes use of this `$movieId` variable, replacing the hard-coded `m01` value from earlier.

To use this query we need to pass our variables along with this dynamic query. In the web browser window, look for the `Query Variables` panel (at the bottom of the left side).

Open this and add our variable as JSON:

```
{
  "movieId": "m01"
}
```

We should now be able to execute our query and see the movie.

Default values

Variables can be given a default value, which can be useful when we want to fall back on one specific result.

Add a default value to our `$movieId` variable by adding `= "m01"` to the definition:

```
query MovieQuery($movieId: String! = "m02") {
  movie(id: $movieId) {
    title
  }
}
```

Now if we delete the query variable from the panel, and run this query, it will display our default movie (movie `m02` in this example).

Adding variables to our queries makes them more flexible and saves us having to mess around with strings when generating the queries. However we can add more logic to our queries. One way to do this is to use `directives`.

Directives

In GraphQL, a directive is a way to add conditional logic to our queries. We can have the query make adjustments on the fly based on conditions (`if something is true, then...`).

We can do this using two directives, `@include` and `@skip`.

Directive: @include

Let's create a query that gives us a list of who stars in a movie only when we want the extra information.

```
query MovieInfo(
  $movieId: String! = "m02",
  $withStars: Boolean = false
) {
  movie(id: $movieId) {
    title
    starring @include(if: $withStars) {
      name
    }
  }
}
```

This brings two changes to the query. First we add another variable, `$withStars` to our named operation. This is an optional boolean with a default value of `false`.

We then use this variable in the directive on the `starring` field. By adding `@include(if: $withStars)` this tells the query to only include this field if `$withStars` is `true`.

Try adding the above query and pass in the variables JSON:

```
{
  "movieId": "m01",
  "withStars": true
}
```

Pressing Play should return a movie with the list of starring actors.

Directive: @skip

It might make more sense to ask our API for a summary of a movie, and expect the full version to contain extra information. This would allow us to expand the definition later without being locked into including specific parts.

Let's use the `@skip` directive and adjust our query to present a summary version.

```
query MovieInfo(
  $movieId: String! = "m02",
  $summary: Boolean = false
) {
  movie(id: $movieId) {
    title
    starring @skip(if: $summary) {
      name
    }
  }
}
```

This example replaces the `@include` with `@skip` and makes use of a variable `$summary`. This will give us the option of being able to request a summary of the movie as needed, but by default will return the extended data.

We can test this by passing in the variables JSON:

```
{
  "movieId": "m01",
  "summary": true
}
```

Try changing these variables and find more ways to adjust the returned data.

Mutations

So far we've only been using the default `query` to fetch data from our API. In REST terms, this is similar to sending a `GET` request, in that it requests data but should not send changes to our server.

A convention in GraphQL is to use `mutation` when we send a query that will make changes to our data. This is more like how we might use a `POST` or `PUT` in REST.

We can use a `mutation` to create a new movie on the server. Before we send a query, we need to set up the server to receive it.

In our `index.js` file we begin by adding two new `type` entries to our `schema` section:

```
type NewMovie {
  id: String!
  title: String!
  starring: [String!]!
}

type Mutation {
  createMovie(
    title: String!,
    starring: [String!]!
  ): NewMovie
}
```

The first part, `NewMovie` is similar to the original `Movie` definition but we've simplified the `starring` field. This will make our server response a little simpler.

We then create a new `Mutation` type. This is where we define our mutation queries. In this case we're defining a `createMovie` query, which returns an object of type `NewMovie`.

Next we update our `resolvers`:

```
const resolvers = {
  Query: {
    actors: () => Object.values(db.actors),
    movies: () => Object.values(db.movies),
    actor: (parent, { id }) => db.actors[id],
    movie: (parent, { id }) => db.movies[id],
    moviesStarring: (parent, { name }) => {
      const actor = Object.values(db.actors).find((actor) =>
actor.name === name);
      return actor.appearsIn;
    }
  },
  Mutation: {
    createMovie: (parent, { title, starring }) => {
      // We could validate and update a database here
      return {
        id: 'new-id',
        title,
        starring
      };
    }
  }
}
```

We now have a `createMovie` resolver. Since we're not working with a real database, we don't have that logic in place but for now we can return a "new" movie to show how the mutation works.

Let's test our mutation:

```
mutation NewMovie(
  $title: String!,
  $starring:[String!]!
) {
  createMovie(
    title: $title,
    starring: $starring
  ) {
    id
    title
    starring
  }
}
```

We have changed the `query` to a `mutation`, and adjusted the variables so that this mutation will expect a `title` and an array of `starring` strings.

When it creates the movie, it will expect an object in response that will include a new `id`, the `title` we gave and the `starring` array.

We can test this using some variables:

```
{
  "title": "A New Movie",
  "starring": ["Actor One", "Actor Two"]
}
```

This now allows us to send new movies to our server. While this is a good start, what if something goes wrong? It would be good to have some way to check what we send and return an error. To do this we can make use of `inline fragments`.

Inline Fragments

Inline Fragments are used when we want to query a field that returns more than one type of result.

We'd like to have a query that can return a `NewMovie` type if all goes well, but also will return an `Error` object if something goes wrong.

To prepare this we need to make a few changes to our `typeDefs`. First we add a new `Error` type:

```
type Error {
  id: String!,
  message: String!
}
```

Then we can make a new `union` type that combines both `NewMovie` and `Error`:

```
union NewMovieOrError = NewMovie | Error
```

A union is a special type that combines two or more types. It's a way of saying this type can be either a `NewMovie` or an `Error`.

To finish the schema changes, update our existing `createMovie` mutation to return an object of type `NewMovieOrError`:

```
type Mutation {
  createMovie(
    title: String!,
    starring: [String!]!
  ): NewMovieOrError
}
```

Next we update our `resolvers`. Let's begin by updating our `createMovie` resolver to return an error if the `title` is empty:

```
createMovie: (parent, { title, starring }) => {
  // We could validate and update a database here
  if (!title.length) {
    return {
      id: 'err-01',
      message: 'Title should not be blank'
    }
  }
  return {
    id: 'new-id',
    title,
    starring
  };
}
```

When GraphQL returns `NewMovieOrError` from the request, we need to tell it what type of object it actually is. To do this we need to resolve the type manually.

This means adding one more resolver, specifically for the `NewMovieOrError` type. Add this to the `resolvers` object:

```
NewMovieOrError: {
  __resolveType(obj, context, info){
    if(obj.message){
      return 'Error';
    }
    if(obj.title){
      return 'NewMovie';
    }
  },
}
```

Now we can try a GraphQL query. If you had any trouble with the above steps, you can refer to the complete index.js file here.

Our query can look something like this:

```
mutation NewMovie(
  $title: String!,
  $starring:[String!]!
) {
  createMovie(
    title: $title,
    starring: $starring
  ) {
    ... on Error {
      id,
      message
    }
    ... on NewMovie {
      id,
      title,
      starring
    }
  }
}
```

You'll notice two additions, `... on Error` and `... on NewMovie`. These allow us to shape a request that depends on the type of object given to us, whether it's a `NewMovie` or an `Error`.

Try it out by setting the variables with an empty title:

```
{
  "title": "",
  "starring": ["Actor One", "Actor Two"]
}
```

This should give us a response like:

```
{
  "data": {
    "createMovie": {
      "id": "err-01",
      "message": "Title should not be blank"
    }
  }
}
```

This is a powerful way to add conditional logic that makes our queries more robust.

Meta fields

When working with a new GraphQL API it can be worthwhile taking a look around to see what sort of queries we can carry out. To help with this, GraphQL gives us a set of special queries. We will take a look at some here.

__schema

Querying the `__schema` field on the root type gives us access to lots of different aspects of the schema. From the full list of `types`, the available `directives`, `queryType` fields and `mutationType` fields.

For example if we wanted to see what sorts of queries (`query` or `mutation`) we can run on our movies database, we can check by running:

```
{
  __schema {
    queryType {
      fields {
        name
      }
    }
    mutationType {
      fields {
        name
      }
    }
  }
}
```

__typename

If we were building a query using `inline fragments` and wanted to know the type name of a movie and actor field, we can use the `__typename` field to obtain it.

```
{
  movie(id:"m01") {
    title,
    starring {
      __typename
      name
      appearsIn {
        title
      }
    }
  }
}
```

This is just some of what you can do. Find out more in the Introspection documentation.

Adding new queries

Before we finish up this section let's add one more query, which might be useful when we want to build a web interface to our API. We'll add an extra query to let us look up a list of movies by actor name, called `moviesStarring`.

Begin by adding the type definition to our `typeDefs`:

```
type MoviesStarring {
  name: String!
  movies: [Movie]!
}

type Query {
  actors: [Actor]
  movies: [Movie]
  actor(id: String!): Actor
  movie(id: String!): Movie
  moviesStarring(name: String!): MoviesStarring
}
```

Here we add a new type called `MoviesStarring`, which expects a name and an array of movies. Then we add one new query to the `Query` block, called `moviesStarring`.

Then add a `resolver` to our `resolvers` object:

```
moviesStarring: (parent, { name }) => {
  const actor = Object.values(db.actors).find((actor) =>
actor.name.toLowerCase() === name.toLowerCase());
  return {
    name,
    movies: actor.appearsIn
  };
}
```

This will let us look up movies by searching for the actor's name. It compares the given value (converted to lowercase) against each actor name. Then if it finds a match, it will return an array of `movie` objects.

We can query it as so:

```
{
  moviesStarring(name:"Steve Buscemi") {
    name
    movies {
      title
    }
  }
}
```

Which should give us a JSON response:

```
{
  "data": {
    "moviesStarring": {
      "name": "Steve Buscemi",
      "movies": [
        {
          "title": "Reservoir Dogs"
        },
        {
          "title": "Pulp Fiction"
        }
      ]
    }
  }
}
```

In this way, it's quite simple to add new ways to access our API and put together useful ways for our API users to make use of it. It's good to think about how people might want to access data while designing a GraphQL schema.

Lets make use of this new query while integrating GraphQL into a web page.

GraphQL from the browser

To show how we can use GraphQL in the browser, we will set up a simple demo. This demo will be a web page that lets our users select an actor, and then show a list of movies this actor has been in.

This will make use of two GraphQL queries, `actors` and `moviesStarring`.

Before we set up our front-end, we need a server.

Remote Apollo server

If you are familiar with setting up a Node.js server, you can deploy our example wherever you like. It can be deployed to Heroku, AWS or your own server.

However to save some time, I have set up the example Movies API on Vercel.

We will be able to use this remote server with a `GET` request to use GraphQL queries from the browser. For example, a request lke this will return the `JSON` response:

```
{
  "data": {
    "movies": [
      {
        "title": "Reservoir Dogs"
      },
      {
        "title": "True Romance"
      },
      {
        "title": "Pulp Fiction"
      }
    ]
  }
}
```

This will allow us to send requests from a website. Let's set that up next.

Website: HTML & CSS

As this is a getting started guide, we will try to keep our example as simple as possible so that we can focus on making use of our GraphQL queries.

If at any point you find yourself stuck, check the source for this finished example.

Let's get started. Make a HTML file (`index.html`) and paste in the following HTML code.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Starring in... Movies API Demo</title>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com"
crossorigin="">
  <link href="https://fonts.googleapis.com/css2?
family=Limelight&amp;family=Open+Sans&amp;display=swap"
rel="stylesheet">
  <link rel="stylesheet" href="https://hop.ie/graphql-apollo-
example/styles.css">
</head>
<body>
  <h1>Starring in...</h1>
  <section id="dropdown">
    <select name="actor">
      <option value="loading">Loading...</option>
    </select>
  </section>
  <section id="info" style="display: none;">
    <p><span id="name"></span> starred in...</p>
    <ul id="movies"></ul>
  </section>
  <script>
    // Our GraphQL logic will go here
  </script>
</body>
</html>
```

Whild developing our local web site we need to serve it and access it on `localhost`. There are many ways to do this but one simple way is to use Python's built-in server.

If you are on a Mac, you can run the local server by navigating in your terminal to the folder containing `index.html` and running:

```
python -m SimpleHTTPServer 8020
```

If you're using Python 3, you might want to try this version:

```
python -m http.server 8020
```

Now you should be able to open `http://localhost:8020/` and see the starting HTML. Let's look at how we can fetch some data from the GraphQL API.

Making a GET request

Now that we have some HTML, and it's running in our browser, we can add some JavaScript to get a feel for how we might use our GraphQL API.

Let's start with a simple query to get a list of movies. Add the following between the `script` tags:

```
// Get the movies from the API
const query = `{
  movies { title }
}`;
const variables = `{}`;
const queryString = `?
query=${encodeURIComponent(query)}&amp;variables=${encodeURIC
omponent(variables)}`;
const url = `https://graphql-apollo-
example.vercel.app/${queryString}`;

async function sendRequest() {
  const response = await fetch(url);
  const data = await response.json();
  console.log(data);
};

sendRequest();
```

In this example we set up some variables, `query` (and an empty `variables` container), then we use these to build a `queryString`. This makes a `GET` request to out server, when we lastly prepend the server address to finish building the `url`.

We use `encodeURIComponent` to convert the GraphQL query, and the `variables` JSON, into a URL-friendly format.

Lastly we set up an `async sendRequest` function. This is where we use the browser's `fetch` method to hit the given URL, process the returned `JSON`, and log it to the console.

With our URL built, refresh the page and check the `console` output. It should give us a response like:

```
{
  data: {
    movies: [
      { title: "Reservoir Dogs" }
      { title: "True Romance" }
      { title: "Pulp Fiction" }
    ]
  }
}
```

This is similar to what we saw in the previous examples, but we now have this response available in the browser.

Note: Since this is a demo, we're not worrying about browser compatibility. If you do want to use `fetch` you might want to also consider a polyfill such as whatwg-fetch.

Let's build on this idea to make something more useful.

Requesting actors and movies

We'll take this simple idea above and set up some handy functions so that we can create a useful little website.

We begin by removing the content of our `script` tags. Then we'll add a re-usable function to make requests:

```
async function makeRequest(query, variables = '{}') {
  const queryString = `?
query=${encodeURIComponent(query)}&amp;variables=${encodeURIC
omponent(variables)}`;
  const url = `https://graphql-apollo-
example.vercel.app/${queryString}`;
  const response = await fetch(url);
  return await response.json();
}
```

We can now add two more functions to load a list of `actors` and then populate the `select`:

```
async function setupActors() {
  const query = `{
    actors {
      name
    }
  }`;
  const parsedResponse = await makeRequest(query);
  const dropdown =
document.querySelector('select[name=actor]');
  addActorsToDropdown(dropdown, parsedResponse.data.actors);
  dropdown.onchange = onActorChange;
};

function addActorsToDropdown(dropdown, actors) {
  dropdown.remove(0); // Remove the loading option
  const option = document.createElement("option");
  option.value = '';
  option.text = 'Select an actor';
  dropdown.appendChild(option);

  for (const actor of actors) {
    const option = document.createElement("option");
    option.value = actor.name;
    option.text = actor.name;
    dropdown.appendChild(option);
  }
}

async function onActorChange(e) {
  // TODO
}
```

These functions make use of our `makeRequest` function, and set up the `select` with the returned list of actors.

We then add a listener for the `onchange` event of this `select`. When the actor name is selected, it will call `onActorChange`.

If we refresh the page now, we should see the `Loading...` text replaced by a list of actors.

Showing the movies

Let's now add some interactivity. Replace the `onActorChange` function above with the following functions:

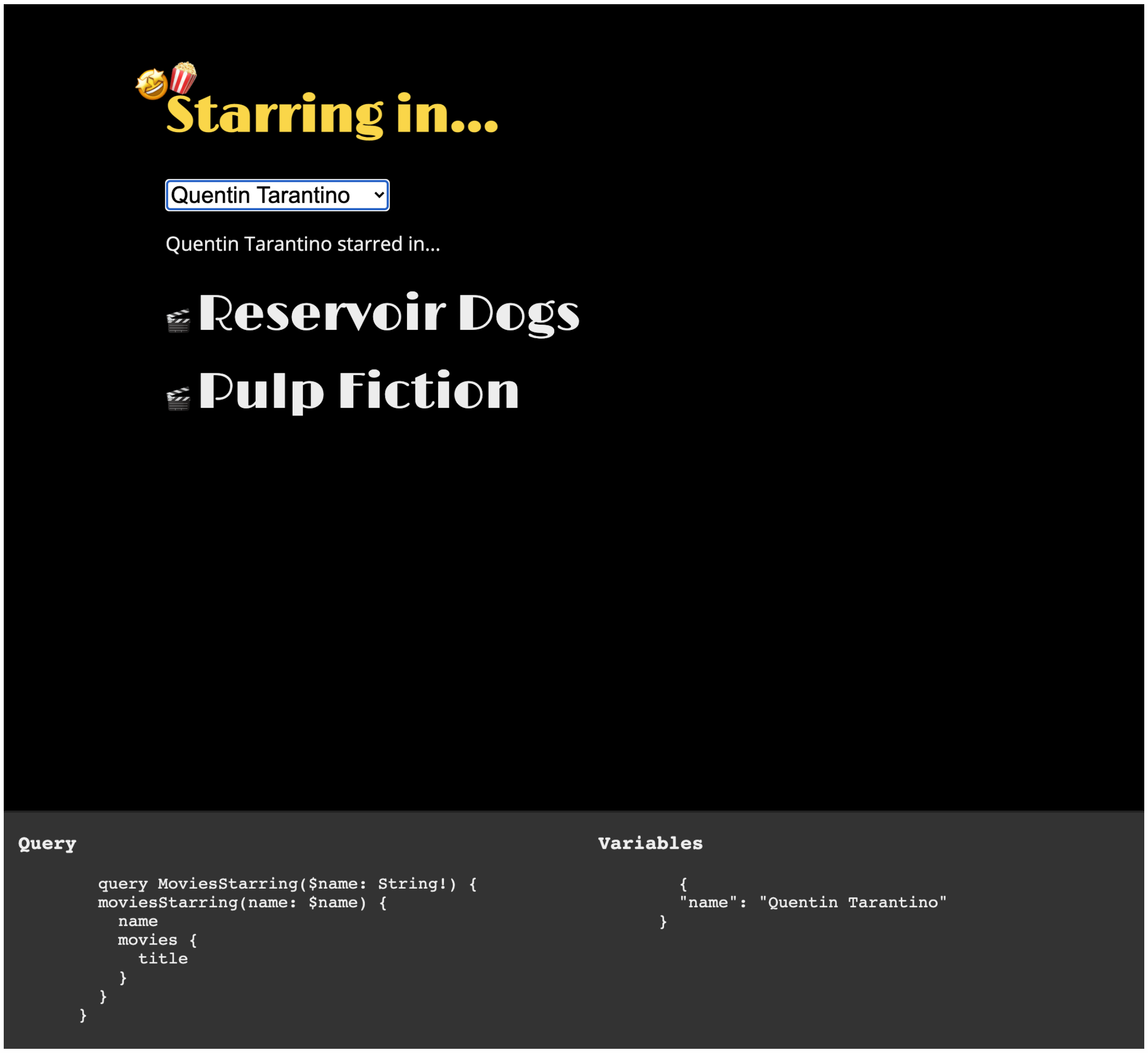
```
async function onActorChange(e) {
  const name = e.target.value;
  const moviesUL = document.querySelector('#movies');
  const nameSpan = document.querySelector('#name');

  // Empty the movies list first
  moviesUL.innerHTML = '';
  // Set the searching-for name
  nameSpan.innerText = name

  if (name === '') {
    showMovieInfo();
    return;
  }

  // Send it to the server
  const query = `query MoviesStarring($name: String!) {
    moviesStarring(name: $name) {
      name
      movies {
        title
      }
    }
  }`;
  const variables = `{
    "name": "${name}"
  }`;
  const parsedResponse = await makeRequest(query, variables);
  const queryResult = parsedResponse.data.moviesStarring;
  showMovieInfo(queryResult.name, queryResult.movies);
}

function showMovieInfo(name, movies) {
  const info = document.querySelector('#info');
  const moviesUL = document.querySelector('#movies');

  if (!name) {
    info.style.display = 'none';
    return;
  }

  for (const movie of movies) {
    const movieLI = document.createElement('li');

movieLI.appendChild(document.createTextNode(movie.title));
    moviesUL.appendChild(movieLI);
  }
  info.style.display = 'block';
}
```

Let's start from the function called by the `onchange` event. Our `onActorChange` function will firstly clear out any existing movies, and set the `name` of the actor we selected.

We then make a request to the server looking for `MoviesStarring`. We set `variables` JSON to include the actor name, then send the result of this query to the `showMovieInfo` function.

The `showMovieInfo` function gets the name and a list of movies, and uses these to built then show the `ul` list.

Now if you refresh the page, you should be able to select a name and see some of the movies they starred in.



Quentin Tarantino starred in...

Reservoir Dogs

Pulp Fiction

Query

```
    query MoviesStarring($name: String!) {
    moviesStarring(name: $name) {
      name
      movies {
        title
      }
    }
  }
```

Variables

```
    {
    "name": "Quentin Tarantino"
  }
```

Showing query information

If you want to see a little extra info, I've added some more logic to the online GraphQL Apollo example, showing the query info to the bottom of the screen. This might be helpful to see the queries as they occur.

Adding mutations

We have shown how to send `query` requests to our GraphQL API, but how can we send `mutation` requests? The answer is to send a `POST`.

Before we wrap up this getting started guide, let's try sending a `mutation` to our Apollo server to show it in action.

Replace the JavaScript in our `script` tags with this snippet:

```
const query = `mutation NewMovie(
  $title: String!,
  $starring:[String!]!
) {
  createMovie(
    title: $title,
    starring: $starring
  ) {
    ... on Error {
      id,
      message
    }
    ... on NewMovie {
      id,
      title,
      starring
    }
  }
}`;

const variables = {
  title: "Test movie",
  starring: ["Actor One", "Actor Two"]
};

async function mutationRequest(query, variables) {
  const url = 'https://graphql-apollo-example.vercel.app';
  const response = await fetch(url, {
    method: 'post',
    headers: {
      'Accept': 'application/json, text/plain, */*',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({query, variables:
JSON.stringify(variables)})
  });
  const responseJSON = await response.json();
  console.log(responseJSON);
}

mutationRequest(query, variables);
```

If we check out `console` we should see a data response that confirms the movie details we sent, along with an `id`. We can test the above with an empty movie title to see the error response too!

This simply demonstrates how the `POST` option can enable sending mutations, but we don't have a form for the page to send info the server. I'll leave that as an exercise for the reaser, should you wish to give it a go.

Wrapping up

By now we should have a good understanding of what GraphQL is, when we might want to use it, and be confident using GraphQL queries for generating requests to our APIs.

We've learned how to build our own GraphQL server, including how to define a schema. Within the schema we have used `types` to define the shape of the data we expect to use along with `query` and `mutation` queries.

We then got our server running and explored how to build all sorts of queries, including variables, inline fragments, named operations and more, and made us of our server with requests from a web site.

Congratulations! I hope you have found this a helpful and fun getting started guide to GraphQL.

Next steps

Hopefully by now you've gained a good working knowledge of what GraphQL is about, and why you might want to use it. You should be able to start thinking of ideas of how you might to create a project to try out your new skills, or look up some blog posts and see what's new with GraphQL.

I certainly recommend installing and setting up some implemtations of GraphQL yourself to find what works for you. If you're using a framework such as React, Vue or Svelte to make apps, you can try integrating GraphQL into your workflow.

You should also make sure to check out more advanced guides to increase your knowledge and find new ways GraphQL can save you time and bandwidth.

GraphQL implementations

No matter what language you want to use, you'll likely find a tool to integrate GraphQL. Check out the official GraphQL Code page for lots of options.

For example, some front-end tools you might want to investigate include:

- Apollo Client
- Relay

- graphql-request
- Lokka

If you're looking for a way to generate powerful yet static websites while making the most of GraphQL, be sure to check out Gatsby.

Best practices and testing

The official GraphQL website has a guide to best practices that includes pagination, caching, and more. You might want to also check out their Thinking in Graphs guide.

When using Apollo Server, you might want to delve into their Integration testing guide to get started. They also offer a Mocking option that helps when constructing tests or using Apollo with Storybook.

Third party services

There is a lot of movement in software-as-a-service world around GraphQL, but a couple of options to begin with would be AWS AppSync, Amazon's tool for deploying GraphQL APIs, and GraphCMS, a content management service built around GraphQL.

Further learning

Learn some of the theory behind GraphQL by studying up on Graph Data Structure in

JavaScript.

When you're investigating GraphQL APIs, be sure to keep this official Introspection guide handy as a reference.

Thank you

Thank you for purchasing this guide. I hope it has helped you get started with GraphQL!

Please be sure to check GetStartedWith.dev for more guides.

If you have feedback or ideas to share, you can reach me anytime at donovan@getstartedwith.dev.

Many thanks,

Donovan Hutchinson
getstartedwith.dev