Jonas Helfer Follow
Software Engineer from Switzerland, now living in San Francisco. GraphQL Evangelist @databricks
Apr 22, 2016 · 13 min read

Tutorial: How to build a GraphQL server

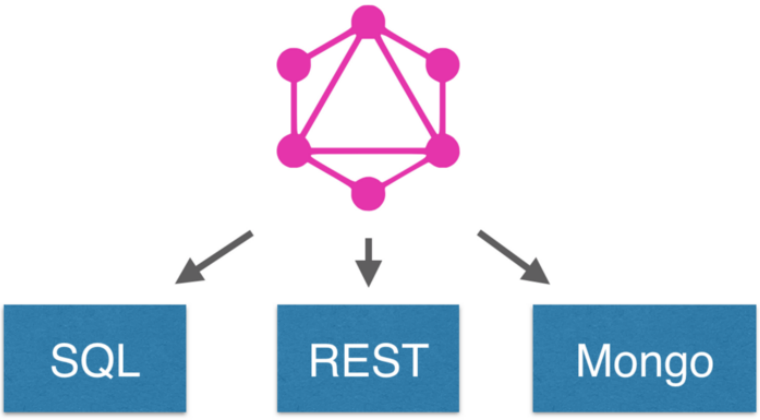
(Updated for 2018!) Talk to MongoDB, SQL and REST with these simple steps

Apollo is a set of tools for building GraphQL applications, especially suited for building on top of your existing data. There are three main components that you can use to build a production-ready GraphQL app:

1. **Apollo Client**: A GraphQL client for every frontend platform.
2. **Apollo Server**: A library for writing GraphQL servers with JavaScript.
3. **Apollo Engine**: A GraphQL gateway that provides caching, error tracking, and performance tracing.

This is a tutorial for the second and third parts—how to build a GraphQL server that connects to multiple backends: a SQL database, a MongoDB database and a REST endpoint. We'll be combining all of them to build a very basic blog with authors, posts and views.

> *Jonas Helfer originally wrote this post in April 2016, but we've updated it since (most recently in December 2017) to make sure it's using the newest Apollo and GraphQL packages available. Huge thanks to Johanna Griffin for adding those updates and thoroughly testing everything!*



The tutorial has nine parts:

1. Setting up
2. Defining a schema
3. Mocking your data
4. Writing resolvers
5. Connecting to a SQL database
6. Connecting to MongoDB
7. Using REST services from GraphQL
8. Setting up performance tracing
9. Setting up result caching

. . .

1. Setting up

This tutorial assumes that you've already set up Node.js and npm for your computer, and that you know how to copy-paste things to the command line. Naturally, you'll also need a text editor of sorts.

To follow along with the MongoDB part (optional), you'll also need a MongoDB server running somewhere.

If you have all of that, there is just a little bit more boilerplate to set up: Babel for ES6 features, and a simple express server. You can do this yourself if you like, but you can also just clone the starter kit for this tutorial:

```
git clone https://github.com/apollographql/apollo-tutorial-
kit
cd apollo-tutorial-kit
npm install
```

This will download the starter kit from GitHub and install all the npm packages you need to get started.

Once the installation is finished, you can launch the server with this command:

```
npm start
```

If all goes well, the server should now print out a message that it is listening on port 8080. If you open localhost:3000/graphiql in your browser, you should now see the GraphiQL UI:

Try typing this basic query and running it.

If it worked, you're all ready to get started writing your GraphQL server!

2. Defining a schema

At the core of any GraphQL server is a schema. The schema defines types and their relationships. It also specifies which queries can be made against the server.

In our example, the schema is defined in `data/schema.js` :

```
import {
  makeExecutableSchema,
  addMockFunctionsToSchema,
} from 'graphql-tools';
import mocks from './mocks'

const typeDefs = `
type Query {
  testString: String
}
`;
```

The most important part of this file is the `typeDefs` string which defines our schema in the GraphQL schema language:

```
type Query {
  testString: String
}
```

The type definitions are compiled to an executable GraphQL schema by the `makeExecutableSchema` function from `graphql-tools`. With the current schema, our server provides exactly one field on the root query type: `testString`, which returns a string.

The schema notation supports all GraphQL types, but in this tutorial we are only going to use a few of them. If you want to know about the others, you can check out the docs for the Apollo graphql-tools package.

Adding types and fields

For our blog app, we now have to modify `schema.js` and add the types we need. You can write those inside the `typeDefs` string.

Author is going to have four fields: `id`, `firstName`, `lastName` and `posts`. `posts` is an array that refers to that author's posts.

```
type Author {
  id: Int
  firstName: String
  lastName: String
  posts: [Post]
}
```

Post is going to have five fields: `id`, `title`, `text`, `views` and `author`. Author refers to the author type we just defined:

```
type Post {
  id: Int
  title: String
  text: String
  views: Int
  author: Author
}
```

Now that we've defined the types, we need to modify the Query type to tell the server about the queries that users are allowed to make:

```
type Query {
  author(firstName: String, lastName: String): Author
  allAuthors: [Author]
  getFortuneCookie: String # we'll use this later
}
```

Your `schema.js` file should now look as follows:

```
import { makeExecutableSchema, addMockFunctionsToSchem
import mocks from './mocks';

const typeDefs = `
type Query {
  author(firstName: String, lastName: String): Author
  allAuthors: [Author]
  getFortuneCookie: String # we'll use this later
}

type Author {
  id: Int
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int
  title: String
```

data/schema.js

3. Mocking your data

If you used the `npm start` command to start the server, it will have updated automatically, so you can go to localhost:3000/graphiql, and run the following query:

```
query {
  author(firstName:"Edmond", lastName: "Jones"){
    firstName
    lastName
    posts{
      title
      views
    }
  }
}
```

You should get a response that looks something like this:

```
{
  "data": {
    "author": {
      "firstName": "It works!"
      "lastName": "It works!"
      "posts": [{
        "title": "It works!"
        "views": 34
      }, {
        "title": "It works!"
        "views": -22
      }]
    }
  }
}
```

Are you wondering how your GraphQL came up with that response, even though we didn't tell it where to get data? It's quite simple actually, we told the server in the starter kit to mock the data, and that's what it's doing.

The data doesn't look very realistic though, so we're going to have to change that. If you're not interested in mocking, you can skip to the next section. If you're not sure whether you should be interested in mocking, you can take a quick look at my post on mocking, which explains the benefits.

Customizing mock data

Alright then, let's change the mocking to make it a bit more realistic. Right now the server is using the same string everywhere. We're going to tell it how to mock a first and last name for Author, and how to mock title, text and views for Post. We'll also tell it that if we queried for a specific author, the author returned should have the name we searched for.

In order to do that, we're going to use an npm package called `casual`, which you should now install by running:

```
npm install casual --save
```

Once `casual` is installed, go ahead and modify the file `data/mock.js` to look like this:

```
import casual from 'casual';

const mocks = {
  String: () => 'It works!',
  Query: () => ({
    author: (root, args) => {
      return { firstName: args.firstName, lastName: ar
    },
  }),
  Author: () => ({ firstName: () => casual.first_name,
```

I won't explain here how it works exactly. At a high level, mocks for each type just define a function that gets run when an object of that type should be returned. You can find more information in the `graphql-tools` docs.

Go ahead and restart your server with `npm start`, then rerun the query you ran earlier. The output should make a lot more sense now.

4. Writing resolvers

So far, we've only set up the schema and tested it with some mock data. In order for the server to be more useful, we have to tell it how to respond to a query. In GraphQL, this is done through resolvers. Every field can have a resolver, which is a function that tells the GraphQL server how to return data for that field, if it appears in a query.

The first resolver for the `author` field on Query is the first one to be called for the example query that you ran earlier. The return value of that resolver then gets passed to the resolvers on the fields of Author, for example, `posts`. The result of that gets passed on to the next resolver, until a scalar type is reached. Scalar types are GraphQL-speak for leaf nodes in the graph: Strings, numbers, booleans, etc.

Let's go ahead and define resolvers for our schema. At the minimum, we need to define a resolver function for each field that either returns a non-scalar type or takes any arguments. We'll define the resolvers in `data/resolvers.js`. Create that file and copy-paste the following:

```
const resolvers = {
  Query: {
    author(root, args) {
      return { id: 1, firstName: 'Hello', lastName: 'W
    },
    allAuthors() {
      return [{ id: 1, firstName: 'Hello', lastName: '
    }
  },
  Author: {
    posts(author) {
      return [
        { id: 1, title: 'A post', text: 'Some text', v
        { id: 2, title: 'Another post', text: 'Some ot
      ];
    }
  }
```

data/resolvers.js

Before you can try the new server, we need to tell it to stop using mock data and call the resolvers instead. Follow the steps below in `schema.js`:

1. Comment out the call to `addMockFunctionsToSchema`
2. Add the resolvers to `makeExecutableSchema({ typeDefs, resolvers })`.
3. Import the `resolvers` object from `./resolvers.js`.

All together, the `schema.js` file should look more like this when you're done:

```
// import mocks from './mocks' <-- comment out
import resolvers from './resolvers';

const typeDefs =`< type Post...etc >`
```

```
// Add resolvers option to this call
const schema = makeExecutableSchema({ typeDefs, resolvers
});

// addMockFunctionsToSchema({ schema, mocks }); <-- comment
out

export default schema;
```

If you run the query from earlier again, you'll see that it returns exactly what the resolvers return. If you see a `null` value instead, make sure you passed the resolvers into `makeExecutableSchema`.

Now that we understand how resolver functions work, let's hook them up to a SQL database!

5. Connecting to SQL

In order to get our data from a SQL database into the GraphQL response, we're going to write a connector, which is an object that encapsulates database logic. Then, we'll use that connector from the resolver function. Technically you could write the database connection logic directly in the resolvers, but it's much better to keep your resolvers extremely simple and separate your concerns. If you need to change backends later, only your connectors have to change, and the resolvers can stay the same.

In this example, we'll use Sequelize—a SQL ORM—to connect to a SQLite database.

First, install both `sequelize` and `sqlite` as well as `lodash`, which we'll use later:

```
npm install --save sequelize sqlite lodash
```

Next, create a file in the data folder called `connectors.js`. In it, we'll set up the database connection and define the schema of our database. To have some data to query, we'll use `casual` again to create some fake data every time the server starts:

```
import Sequelize from 'sequelize';
import casual from 'casual';
import _ from 'lodash';

const db = new Sequelize('blog', null, null, {
  dialect: 'sqlite',
  storage: './blog.sqlite',
});

const AuthorModel = db.define('author', {
  firstName: { type: Sequelize.STRING },
  lastName: { type: Sequelize.STRING },
});

const PostModel = db.define('post', {
  title: { type: Sequelize.STRING },
  text: { type: Sequelize.STRING },
});

AuthorModel.hasMany(PostModel);
PostModel.belongsTo(AuthorModel);

// create mock data with a seed, so we always get the
casual.seed(123);
db.sync({ force: true }).then(() => {
  _.times(10, () => {
    return AuthorModel.create({
```

Now that we have the connectors, we have to modify our `resolvers` file to import and use the connector we just defined. It's pretty straightforward:

```
import { Author, View } from './connectors';

const resolvers = {
  Query: {
    author(_, args) {
      return Author.find({ where: args });
    },
    allAuthors(_, args) {
      return Author.findAll();
    }
  },
  Author: {
    posts(author) {
      return author.getPosts();
    }
  },
  Post: {
    author(post) {
```

data/resolvers.js

Go ahead and run the following query in your GraphiQL window:

```
{
  author(firstName: "Edmond", lastName: "Jones") {
    firstName
    lastName
    posts {
      title
    }
  }
}
```

You should get the following output:

```
{
  "data": {
    "author": {
      "firstName": "Edmond",
      "lastName": "Jones",
      "posts": [
        {
          "title": "A post by Edmond",
        }
      ]
    }
```

```
    }
  }
```

That wasn't hard at all, right? Let's add MongoDB to the mix.

6. Connecting to MongoDB

Now I want to show you one of the greatest features of GraphQL: joining across different backends, in this case SQL and MongoDB. We'll use this to display view counts for our posts.

Of course, we could have used the SQL database for the views as well, but often you have to work with existing legacy backends, or you have certain scaling requirements that can only be met by splitting your backend into different services. To illustrate how useful Apollo and GraphQL can be in those cases, we're going to use MongoDB to store the views.

For this next part to work, you need to have MongoDB installed.

> *On macOS, run* `brew install mongodb`*, create the directory for your data to be stored in with* `mkdir -p /data/db`*, and correct the permissions on this directory if need be with* `chown -R <user> /data/db`*. Read the MongoDB docs for directions on other operating systems.*

Don't forget to start the database:

```
mongod
```

Feel free to just skim the following code or skip to the next section if you'd prefer not to install MongoDB.

. . .

Still here? Great! Let's get MongoDB wired up to our GraphQL schema. As you probably guessed, we first need to define a connector. By now you'll probably be able to figure out where each part should go. Make sure you don't forget to install mongoose!

```
npm install --save mongoose
```

Add the following code to `connectors.js`:

```
// at the top with imports:
import Mongoose from 'mongoose';

// somewhere in the middle:
Mongoose.Promise = global.Promise;

const mongo = Mongoose.connect('mongodb://localhost/vi
  useMongoClient: true
});

const ViewSchema = Mongoose.Schema({
  postId: Number,
  views: Number,
});

const View = Mongoose.model('views', ViewSchema);

// modify the mock data creation to also create some v
casual.seed(123);
db.sync({ force: true }).then(() => {
  _.times(10, () => {
    return AuthorModel.create({
      firstName: casual.first_name,
      lastName: casual.last_name,
    }).then((author) => {
      return author.createPost({
        title: `A post by ${author.firstName}`
```

data/connectors.js

In `resolvers.js`, some of our resolvers are already calling the MongoDB connector, like so:

```
import { Author, View } from './connectors';
```

```
  Post: {
    author(post) {
      return post.getAuthor();
```

```
    },
    views(post) {
      return View.findOne({ postId: post.id })
             .then((view) => view.views);
    },
  },
```

Start your MongoDB database on the CLI with `mongod`, and run the original query again now:

```
{
  author(firstName: "Edmond", lastName: "Jones") {
    firstName
    lastName
    posts {
      title
      views
    }
  }
}
```

Views should now be pulled from MongoDB and no longer be null.

7. Querying a REST endpoint

Finally, let's also add a REST endpoint to the mix.

GraphQL resolvers can either directly return a value, or they can return a promise. You may not have realized it, but we've been taking advantage of that all along: Both sequelize and mongoose return promises!

One thing that these promises let us do is get data from other services. I couldn't really think of a great example API to include here, so I ended up choosing the fortune cookie API. It doesn't have anything to do with the rest of the example, but it's good enough to illustrate the concept.

First, you'll have to install an npm package called `node-fetch`.

```
npm install --save node-fetch
```

Once that's done, add the following to `connectors.js`:

```
// add this import at the top
import fetch from 'node-fetch';

// add this somewhere in the middle
const FortuneCookie = {
  getOne() {
    return fetch('http://fortunecookieapi.herokuapp.co
      .then(res => res.json())
      .then(res => {
        return res[0].fortune.message;
      });
```

Now that you have the connector, I'm sure you can figure out how to modify `resolvers.js` to use the connector for the `getFortuneCookie` field! To see if you did it right, you can run the following query:

```
{
  author(firstName: "Edmond", lastName: "Jones"){
    firstName
    lastName
  }
  getFortuneCookie
}
```

You should get a response that looks similar to this:

```
{
  "data": {
    "author": {
      "firstName": "Edmond",
      "lastName": "Jones"
    },
    "getFortuneCookie": "You can't unscramble a scrambled
egg."
  }
}
```

Did it work? If so, congratulations, you've just written a GraphQL server that uses a SQL, MongoDB and REST!!!

If it didn't quite work, you can always take a look at the server-tutorial-solution branch and compare it to your files.

8. Setting up performance tracing

Now, if you're interested in seeing your server in action and visualize how your queries are being executed, you can use GraphQL tracing with Apollo Engine. With Engine, you'll be able to see all requests that your server gets and all errors that are thrown. You'll also be able to see performance traces for specific queries and, in the next step, turn on query result caching.

Adding Engine to your server

To get started, you'll need to install the `apollo-engine-js` npm package in your server, and the `compression` middleware to make things more efficient:

```
npm install --save apollo-engine compression
```

Now we need to turn on tracing. Thankfully this is super easy—just pass one more option to your Apollo Server middleware and add `compression` to `server.js`, like so:

```
import compression from 'compression';

graphQLServer.use(compression());

graphQLServer.use('/graphql', bodyParser.json(),
graphqlExpress({
  schema,

  // This option turns on tracing
  tracing: true
}));
```

To set up Apollo Engine, which is going to collect those traces and display them in a nice UI, it's just a few more lines of code:

```
import { Engine } from 'apollo-engine';

const GRAPHQL_PORT = 3000;
const ENGINE_API_KEY = 'API_KEY_HERE'; // TODO
```

```
const engine = new Engine({
  engineConfig: {
    apiKey: ENGINE_API_KEY,
  },
  graphqlPort: GRAPHQL_PORT
});

engine.start();

// This must be the first middleware
graphQLServer.use(engine.expressMiddleware());

// Down here is compression and graphqlExpress
```

Take note that the Engine middleware must be the first one in the stack since it needs to process the raw requests before they've been modified by any other middleware.

Last but not least, you need to get an Engine API key. Do this by logging into the Engine UI at engine.apollographql.com, and following the instructions to add a service.

Once you have your API key, replace our placeholder `API_KEY_HERE` text from above, then run `npm start` to "start your Engine". 😁

Congratulations, you're done with the server part! 🎉 If you made it this far, your `server.js` file should now look something like this:

```
import express from 'express';
import { graphqlExpress, graphiqlExpress } from 'apoll
import bodyParser from 'body-parser';
import compression from 'compression';
import { Engine } from 'apollo-engine';

import schema from './data/schema';

const GRAPHQL_PORT = 3000;
const ENGINE_API_KEY = 'API_KEY_HERE'; // TODO

const graphQLServer = express();

const engine = new Engine({
  engineConfig: {
    apiKey: ENGINE_API_KEY
  },
  graphqlPort: GRAPHQL_PORT
});

engine.start();

// This must be the first middleware
graphQLServer.use(engine.expressMiddleware());
graphQLServer.use(compression());
graphQLServer.use(
  '/graphql',
```

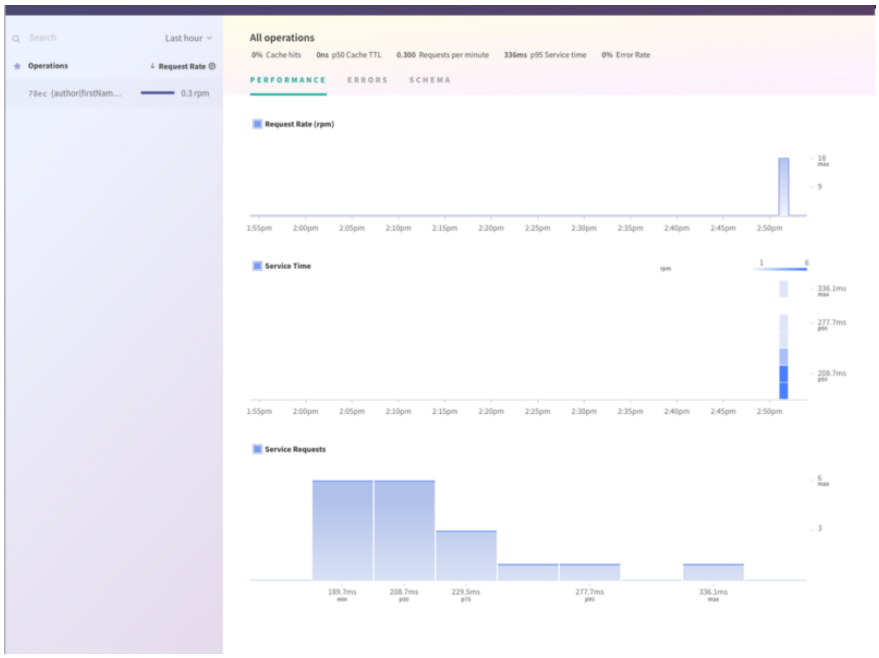
Now, let's go look at our GraphQL server in the Engine UI!

Viewing your data in the UI

At this point if you run the following query a few times against your server, in a few seconds you'll start seeing basic query performance data in the Engine UI under the service you just created:

```
{
  author(firstName: "Edmond", lastName: "Jones") {
    firstName
    lastName
  }
}
```

You should see something similar to screenshot below:



Engine UI

To learn about other configuration options, check out the Engine docs.

9. Advanced topic: Setting up caching

Caching speeds up your GraphQL query performance by avoiding actually running queries when you already have the result. Here, we'll configure Engine to cache some of our queries.

First, set cacheControl to true in your Apollo Server options to turn on the Apollo Cache Control feature, just like you did with tracing:

```
graphQLServer.use('/graphql', bodyParser.json(),
graphqlExpress({
  schema,
  tracing: true,
  cacheControl: true
}));
```

Next, tell Engine what to cache by setting a cache policy in your `schema.js` file. In this example, we'll add a cache control hint in the schema using the `@cacheControl` decorator. Let's cache the `getFortuneCookie` field with a `maxAge` of 5 seconds.

```
type Query {
  author(firstName: String, lastName: String): Author
  allAuthors: [Author]
  getFortuneCookie: String @cacheControl(maxAge: 5)
}
```

The last thing we need to do to turn on caching is configure a store in the Engine proxy by adding `stores` and `queryCache` to your Engine configuration object in `server.js`:

```
// The full new config
const engine = new Engine({
  engineConfig: {
    apiKey: ENGINE_API_KEY,
    stores: [
      {
        name: 'inMemEmbeddedCache',
        inMemory: {
          cacheSize: 20971520 // 20 MB
        }
      }
    ],
    queryCache: {
      publicFullQueryStore: 'inMemEmbeddedCache'
    }
  },
  graphqlPort: GRAPHQL_PORT
});
```

Learn more about advanced configurations in the Engine caching documentation. In this case we're using a simple embedded cache, but if you're doing serious production caching you'll want to plug into Memcache.

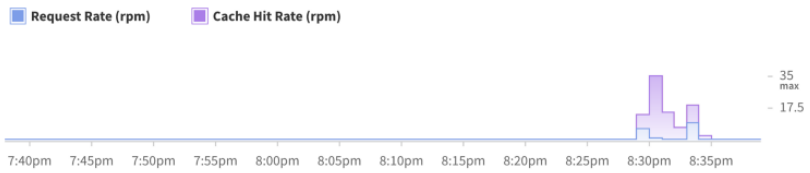
See caching work!

Run a few `getFortuneCookie` queries in GraphiQL:

```
query FortuneCookieQuery {
  getFortuneCookie
}
```

First of all, you should be able to see the responses come back much faster, and now you'll get a new fortune cookie message only every 5 seconds rather than every time.

Then, in a few seconds, you'll start seeing your cache hit rate in the Engine UI. Make sure you browse to the right operation using the list on the left. The view will look similar to:



Note: In the dropdown menu, you can also sort your queries by Cache Hit (%)

Do you see your cache hits? If so, congratulations! You've not only successfully set up a GraphQL server, but you've also configured server-side caching!!! 🎉

. . .

Conclusion

I hope you've enjoyed this tutorial! If you ran into any issues, please let us know in the comments or file an issue on the apollo-tutorial-kit repository!

This tutorial has covered the basics for several topics, but if you want to build a production-ready GraphQL server with Apollo, you'll also have to consider things like authentication, authorization, logging, input validation, and more. We'll go over these things in future articles.

If you're also interested in using GraphQL inside your React components, my new Full-stack React + GraphQL Tutorial might just be what you're looking for, so make sure to check it out!

. . .

> *If you liked this tutorial and want to keep learning about Apollo and GraphQL, make sure to click the "Follow" button below, and follow us on Twitter at @apollographql and @helferjs.*