An Overview of GraphQL

A New Paradigm for Building APIs

BY WILLIAM LYON

CONTENTS

INTRODUCTION TO GRAPHQL

GraphQL is a powerful new tool for building APIs that allows clients (typically web and mobile apps) to ask for only the data they need. Originally designed at Facebook to minimize data sent over the wire and reduce the number of roundtrip API requests for rendering views in native mobile apps, GraphQL has since been open sourced to a healthy community that is building developer-friendly tools. GraphQL has quickly been adopted by several notable companies, including Github, Yelp, Coursera, and Shopify, replacing existing REST APIs. The goal of this Refcard is to introduce you to the concepts of GraphQL, from writing GraphQL queries to building a GraphQL server.

WHAT IS GRAPHQL?

Despite what the name seems to imply, GraphQL is not per se a query language for graph databases — it is instead a query language and runtime for building APIs. The "graph" component of the name comes from the data model that GraphQL uses to work with application domain data, making the observation that your application data is a graph. Connected entities is the natural way that we intuitively think of our data, and GraphQL allows us to express our data in the same way we think about it, as a graph. GraphQL itself is simply a specification, and there are many great tools and libraries available for building GraphQL APIs in almost every language, as well as clients to simplify the process of efficiently querying GraphQL services. The graphql-js reference implementation is the standard GraphQL service implementation.

BENEFITS OF GRAPHQL

- **Schema/type system.** GraphQL requires a user-defined schema built using a strict type system. This schema acts as a blueprint for the data handled by the API, allowing developers to know exactly what kind of data is available and in what format. Developer tools can take advantage of the schema through the use of introspection, enabling documentation, query auto-complete and mocking tools.
- **Request only data needed.** Since GraphQL queries can define what data exactly is needed at query time, all the data required to render an application view can be requested in a single request, reducing the number of roundtrip network requests and with that, the application latency.
- **Wrapping existing data services.** GraphQL can work with any database or data layer, allowing for a single GraphQL service to fetch data from multiple sources in a single request served by the composed API.

A SIMPLE GRAPHQL EXAMPLE

***Note:** All code referenced in this refcard is available online as an Apollo Launchpad, an interactive playground for building and querying GraphQL APIs, here.*

Consider a simple GraphQL schema:

```
type Movie {
  id: ID!
  title: String
  year: Int
  plot: String
  genres: [String]
}

type Query {
  moviesByTitle(title: String!, limit: Int!): [Movie]
}
```

Here we define a `Movie` type, the fields that exist for `Movie`, and a query, `moviesByTitle`, that takes two arguments: `title` and `limit`, both of which are required (indicated by the exclamation point). The `moviesByTitle` query returns an array of `Movie` objects.

Now let's write a simple query to search for movies that contain the string "Matrix":

```
{
  moviesByTitle(title: "Matrix", limit: 3) {
    title
    year
  }
}
```

Breaking down the query:

COMPONENT	DESCRIPTION	EXAMPLE
Named query	Our entry point into the GraphQL API	`moviesByTitle`
Arguments	Arguments can be passed to a GraphQL query	`title: "Matrix",` `limit: 3`
Selection set	Each field to be returned is specified in the query	`title` `year`

(Developers)-[:LOVE]-(Neo4j)

Neo4j is the #1 database for connected data, with native graph storage and processing.

Its property graph model and Cypher query language make it easy to use.

Fast. Natural. Fun.

TRY NEO4J NOW

neo4j

By specifying the fields we want returned, this query is traversing our application data graph to find the data requested.

And the result of our query:

```
{
  "data": {
    "moviesByTitle": [
      {
        "title": "Matrix, The",
        "year": 1999
      },
      {
        "title": "Matrix Reloaded, The",
        "year": 2003
      },
      {
        "title": "Matrix Revolutions, The",
        "year": 2003
      }
    ]
  }
}
```

Note that we haven't covered how to actually fetch this data from our data layer — that logic is implemented in our GraphQL service's resolver functions that handle the individual fields of our query. We'll see how to build resolvers in the next section, but first let's look at GraphQL schemas and the type system in more detail.

GRAPHQL SCHEMA AND TYPES

Building a GraphQL service requires a schema to be defined that describes the types, fields, and queries available for the service. This schema will define the types and GraphQL queries that we'll be able to use in our API. You can think of the schema as the underlying API blueprint or specification. GraphQL services can be implemented in any language, so we need a language-agnostic way to define our GraphQL schema. We use the GraphQL schema language (also called the GraphQL Interface Description Language or IDL) to define our schema and types.

TYPE	DESCRIPTION	EXAMPLE
Object types	An object type that can be fetched, including what fields it provides. For example, `Movie` is an object type, with fields `title`, `year`, `genres`, and `plot`. `[Movie]` indicates an array of `Movie` objects. `!` indicates a required field.	`type Movie { id: ID! title: String year: Int plot: String poster: String genres: [String] similar: [Movie] rating: RATING actors: [Actor] avgStars: Float }`
Scalar types	Common primitive types. By default `Int`, `Float`, `String`, `Boolean`, `ID` (serialized as a string but not necessarily to be human readable).	`title: String`
Query types	An entry point into a GraphQL service. The `Query` type defines the queries available for the GraphQL service.	`type Query { moviesByTitle(title: String!, limit: Int= 3): [Movie] }`
Mutation types	Also an entry point into the GraphQL service, mutations are update operations.	`type Mutation { reviewMovie(review: ReviewInput):UserReview }`
Enumeration types	A type of scalar restricted to a set of pre-defined values.	`enum RATING { G PG PG13 R }`
Interface types	Abstract type that includes a set of fields that must be implemented by another type. Interface types are used when returning an object that can be one of several types.	`interface Person { id: ID! name: String }`
Union types	Similar to interface types, however Union types don't specify common fields.	`union PersonResult = User | Actor`
Input types	Used to pass complex objects to a mutation or query.	`input ReviewInput { rating: Int! movieId: ID! userId: ID! }`

GRAPHQL QUERIES AND MUTATIONS

Queries and mutations are the entry point to the GraphQL service.

FIELDS

Each type is composed of one or more fields, which can be scalar values or complex objects that refer to other types, resulting in nesting. All GraphQL queries must at least specify fields to be returned. This is referred to as the *selection set*.

Consider the following schema:

```
type Query {
  topRatedMovies: [Movie]
}

type Movie {
  id: ID!
  title: String
  year: Int
  plot: String
  poster: String
  imdbRating: Float
  genres: [String]
  similar(limit: Int = 3): [Movie]
  rating: RATING
  actors: [Actor]
  avgStars: Float
}
```

We can query:

QUERY

```
{
  topRatedMovies {
    title
    year
    avgStars
  }
}
```

RESULT

```
{
 "data": {
   "topRatedMovies": [
     {
       "title": "Godfather, The",
       "year": 1972,
       "avgStars": 4.4875
     },
     {
       "title": "Shawshank Redemption, The",
       "year": 1994,
       "avgStars": 4.487138263665597
     },
     {
       "title": "On the Waterfront",
       "year": 1954,
       "avgStars": 4.448275862068966
     }
   ]
 }
}
```

Optionally, we can include the operation type and operation name in our query, in addition to our selection set.

```
query TopRatedMoviesAndRating {
  topRatedMovies {
    title
    year
    avgStars
  }
}
```

This blog post contains a good overview of GraphQL terminology, including terms like *selection set*, *operation*, and *operation name*.

ARGUMENTS

We can also pass arguments to our queries. Arguments can be required or optional. When using an optional argument, we declare a default value. We've seen an example of passing an argument already, but here we also pass an argument to a field on the `Movie` type as well as to our root level query:

```
type Query {
  moviesByTitle(title: String!, limit: Int = 3): [Movie]
}
```

QUERY

```
{
  moviesByTitle(title: "Matrix Reloaded") {
    title
     similar(limit: 2) {
      title
      year
    }
  }
}
```

RESULT

```
{
  "data": {
    "moviesByTitle": [
      {
        "title": "Matrix Reloaded, The",
        "similar": [
          {
            "title": "Matrix, The",
            "year": 1999
          },
          {
            "title": "Lord of the Rings: The Fellowship of
             the Ring, The",
            "year": 2001
          }
        ]
      }
    ]
  }
}
```

VARIABLES

To avoid injection attacks by string interpolation to build queries from dynamic or user-supplied content, we can use variables in our GraphQL query. To use variables, we first declare `$varName` as a valid variable for our query, then replace the value in the query with `$varName`. Finally, we pass the key-value pair `varName: value` in a dictionary alongside the query. For example:

QUERY

```
query MoviesByTitle($movieTitle: String!, $similarLimit: Int) {
  moviesByTitle(title: $movieTitle) {
    title
    similar(limit: $similarLimit) {
      title
      year
    }
  }
}
```

VARIABLES

```
{
 "movieTitle": "Matrix Reloaded",
 "similarLimit": 2
}
```

RESULT

```
{
  "data": {
    "moviesByTitle": [
      {
        "title": "Matrix Reloaded, The",
        "similar": [
          {
            "title": "Matrix, The",
            "year": 1999
          },
          {
            "title": "Lord of the Rings: The Fellowship of
             the Ring, The",
            "year": 2001
          }
        ]
      }
    ]
  }
}
```

MUTATIONS

While queries allow us to request data, GraphQL mutations are used for updating data and provide an additional entry-point into our GraphQL API. To use mutations, we include a `Mutation` type in our schema, which defines the mutations allowed in our GraphQL service. One way to think of mutations is as user-defined commands where the logic is defined in the GraphQL server implementation.

Consider a `UserReview` type, which is composed of a `User`, a `Movie`, and an integer rating of the movie:

```
type UserReview {
  user: User
  rating: Int
  movie: Movie
}
```

To create a new `UserReview`, we must define a mutation to perform this update. Our mutation will need to accept the `userId` and `movieId` arguments as well as the integer `rating`:

```
type Mutation {
 reviewMovie(userId: ID!, movieId: ID!, rating: Int): UserReview
}
```

Note that our mutation returns a `UserReview` object. This means that we can access any of the fields available on `Movie` and `User` (in a nested fashion):

QUERY

```
mutation {
  reviewMovie(userId: "20", movieId: "16", rating: 5) {
    movie {
      title
      similar(limit: 2) {
        title
      }
    }
    user {
      name
    }
    rating
  }
}
```

RESULT

```
{
  "data": {
    "reviewMovie": {
      "movie": {
        "title": "Casino",
        "similar": [
          {
            "title": "Four Brothers"
          },
          {
            "title": "Night and the City"
          }
        ]
      },
      "user": {
        "name": "Nicole Ramsey"
      },
      "rating": 5
    }
  }
}
```

INPUT TYPES

In the previous mutation example, we passed three individual arguments of primitives to define the update we wanted to make. We can instead use an *input type* to pass a more complex object as a variable. Input types are especially useful for mutations where we want to pass our update as a single object.

```
input ReviewInput {
  rating: Int!
  movieId: ID!
  userId: ID!
}
```

Then, we modify our `reviewMovie` mutation to accept an argument of type `ReviewInput`:

```
type Mutation {
 reviewMovie(review: ReviewInput!): UserReview
}
```

Then our query becomes:

QUERY

```
mutation CreateReviewForMovie ($review: ReviewInput) {
  reviewMovie (review: $review) {
    movie {
      title
    }
    user {
      name
    }
    rating
  }
}
```

VARIABLES

```
{
  "review": {
    "movieId":"16",
    "userId":"20",
    "rating": 5
  }
}
```

RESULT

```
{
  "data": {
    "reviewMovie": {
      "movie": {
        "title": "Casino"
      },
      "user": {
        "name": "Nicole Ramsey"
      },
      "rating":5
    }
  }
}
```

FRAGMENTS

A fragment is a reusable set of fields that we can define and reference by name in a GraphQL query. Fragments allow us to avoid repetition in our queries by giving us a way to reference this set of fields by name. To apply a fragment inside a query, we use a fragment spread operator inside our selection set:

QUERY

```
{
  moviesByTitle(title: "Matrix Reloaded") {
    ...movieSummaryFields
  }
}

fragment movieSummaryFields on Movie {
  title
  year
  imdbRating
}
```

RESULT

```
{
  "data": {
    "moviesByTitle": [
      {
        "title": "Matrix Reloaded, The",
        "year": 2003,
        "imdbRating": 7.2
      }
    ]
  }
}
```

INLINE FRAGMENTS

Let's say that we have a union type `PersonResult` returned by a query called `personByName`:

```
type Query {
  personByName(name: String!): [PersonResult]
}

union PersonResult = User | Actor
```

`PersonResult` can be either a `User` or `Actor`. In this case, we'll need to use an *inline fragment* to ensure the result of our query is resolved to the correct type.

QUERY

```
{
  personByName(name: "Tom Cruise", limit:3) {
    ... on Actor {
      name
      movies {
        title
      }
    }
    ... on User {
      name
    }
  }
}
```

RESULT

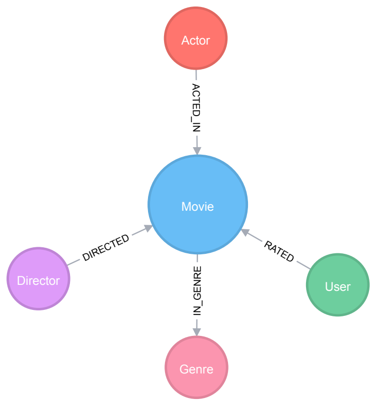
```
{
  "data": {
    "personByName": [
      {
        "name": "Tom Cruise",
        "movies": [
          {
            "title": "Risky Business"
          },
          {
            "title": "Cocktail"
          },
          {
            "title": "Top Gun"
          }
        ]
      }
    ]
  }
}
```

BUILDING A GRAPHQL SERVICE

GraphQL services can be built in almost any language. In this section, we'll make use of the graphql-tools library to build a simple GraphQL API in JavaScript that queries a Neo4j graph database for movies and movie recommendations.

GraphQL, by design, can work with any database (relational or NoSQL) or backend system, but in this example, we'll be using the Neo4j graph database as our persistence layer. Why use a graph database with GraphQL? The idea of application data as a graph is an underpinning design choice of GraphQL. GraphQL enables developers to translate their backend data into the application data graph on the frontend, but if we use a graph database on the backend, we can do away with this impedance mismatch and have graphs all the way down.

First of all, we'll need a Neo4j database (with data) for our GraphQL server to query. For this example, we'll make use of a Neo4j Sandbox instance. Neo4j Sandbox allows us to quickly spin up a hosted Neo4j instance, optionally with existing datasets focused around specific use cases. We'll use the Recommendations Neo4j Sandbox, which includes data about movies and movie reviews and is designed to generate personalized recommendations (for example, by using collaborative filtering to recommend movies based on similar users' reviews).



The graph database model that our GraphQL service will be querying.

We'll make use of the JavaScript driver for Neo4j to connect to and query Neo4j using Cypher, the query language for graph databases.

GRAPHQL FIRST DEVELOPMENT

We'll follow the "GraphQL First" development paradigm. In this approach, we start by defining a GraphQL schema. This schema defines the types and queries available in our API and then becomes the specification for the API. If we were building a complete application, the frontend developers could use this schema to build out the UI while the backend team builds the GraphQL server implementation in parallel, speeding up development. Once we've defined our schema, we'll need to create resolver functions that are responsible for fetching our data, in this case from Neo4j.

GraphQL First Development:

1. Define GraphQL schema
2. Build UI and backend in parallel
3. Deploy

SCHEMA

We first define our GraphQL schema as a string using the GraphQL schema syntax. Our schema will consist of the examples we've used so far:

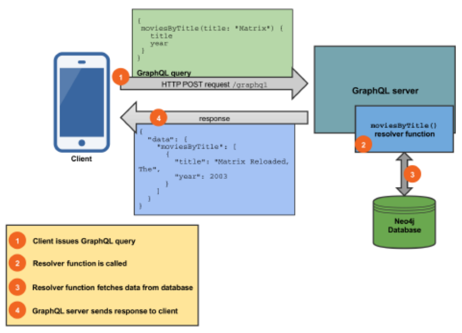
```
// Simple Movie schema
const typeDefs = `
type Movie {
  id: ID!
  title: String
  year: Int
  plot: String
  poster: String
  imdbRating: Float
  genres: [String]
  similar(limit: Int = 3): [Movie]
  rating: RATING
  actors: [Actor]
}
interface Person {
  id: ID!
  name: String
}
type Actor implements Person {
  id: ID!
  name: String
  movies: [Movie]
}
type User implements Person {
  id: ID!
  name: String
  reviews: [UserReview]
}
type UserReview {
  user: User
  rating: Int
  movie: Movie
}
union PersonResult = User | Actor

type Query {
  moviesByTitle(title: String!, limit: Int = 3): [Movie]
  personByName(name: String!, limit: Int!): [PersonResult]
  topRatedMovies: [Movie]
}
type Mutation {
  reviewMovie(review: ReviewInput): UserReview
}
input ReviewInput {
  rating: Int!
  movieId: ID!
  userId: ID!
}
enum RATING {
 G
 PG
 PG13
 R
}
`;
```

***Note**: This schema omits a few examples; see the full code here.*

FETCHING DATA WITH RESOLVERS

After defining a schema, the next step in building a GraphQL server is to specify how to fetch data from our data layer. Each field on each type is resolved by a resolver function in the server implementation. The *resolver function* defines how data is fetched for that field. Resolvers often query databases, other APIs, or data services. This design allows resolvers to make use of database clients, network libraries, or any sort of custom data fetching logic. In this case, we'll query a Neo4j database using the Cypher query language and the JavaScript driver for Neo4j.



```
{
 moviesByTitle(title: "Matrix") {
   title
   year
 }
}
```

```
{
  "data": {
    "moviesByTitle": [
      {
        "title": "Matrix Reloaded,
The",
        "year": 2003
      }
    ]
  }
}
```

1. Client issues GraphQL query
2. Resolver function is called
3. Resolver function fetches data from database
4. GraphQL server sends response to client

Each GraphQL server provides a root level entry point to the GraphQL API, called a *Query* type. Starting from the root level entry point, inside our resolver functions, we fetch the relevant data from our data layer.

Consider this example:

```
Query: {
    // fetch movies by title substring
    moviesByTitle(_, params, context) {
    let session = context.driver.session();
    let query = `
        MATCH (movie:Movie)
        WHERE movie.title CONTAINS $title
        RETURN movie LIMIT $limit;`

    return session.run(query, params)
        .then( result => {
            return result.records.map(record => {
                return record.get("movie").properties })})
}
```

Each resolver is passed these arguments:

- `obj` - the previous object being resolved. Since this is the root level entry point, it is irrelevant for this resolver
- `params` - the arguments provided to the field in the GraphQL query
- `context` - information like the current authenticated user or database connection

The `moviesByTitle` resolver function returns an object of key-value pairs retrieved from the database. We said previously that each field on each type must be resolved by a resolver function. Most GraphQL implementations allow for skipping trivial resolvers — fields where we have scalar values. Some fields on `Movie` are more complex and must be resolved by calls to further resolver functions:

```
// trivial resolvers for Movie type can be omitted
Movie: {
    // find similar movies
    // for users who review this movie
    // what other movies do they review?
    similar(movie, params, ctx) {
      let session = ctx.driver.session();
      params['movieId'] = movie.movieId;
      let query = `
            MATCH (m:Movie {movieId: $movieId})
            MATCH
              (m)<-[:RATED]-(:User)-[:RATED]->(similar:Movie)
            WITH similar, COUNT(*) AS score
            RETURN similar AS movie
            ORDER BY score DESC LIMIT $limit;`;

      return session.run(query, params)
         .then( result => {
            return result.records.map(record => {
               return record.get("movie").properties })})
      },
    avgStars(movie, params, ctx) {
        let session = ctx.driver.session();
        params['movieId'] = movie.movieId;
        let query = `
          MATCH (m:Movie {movieId: $movieId})
          MATCH (m)<-[r:RATED]-(:User)
          RETURN avg(r.rating) AS avgStars`;
          return session.run(query, params)
              .then(result => {
                  return result.records[0].get("avgStars") })
    }
```

***Note** that we've omitted several resolver functions required by our schema. The full code example is available as an Apollo Launchpad here.*

SERVING A GRAPHQL ENDPOINT

Now that we've defined the GraphQL schema and resolver functions, we are ready to serve the GraphQL endpoint. While the GraphQL specification does not specify a client-server protocol for GraphQL, HTTP is the most common choice. A GraphQL server handles a single endpoint, usually `/graphql`, and all requests for the service are directed to this single endpoint. A GraphQL HTTP server should handle both GET and POST requests. Most of the common GraphQL server libraries, such as graphql-express (a common library for serving a GraphQL endpoint using the Express Node.js webserver), provide this behavior.

To use graphql-express to serve our GraphQL endpoint, we use the `makeExecutableSchema` function from graphql-tools and pass the result to the `graphqlExpress` function from graphql-express:

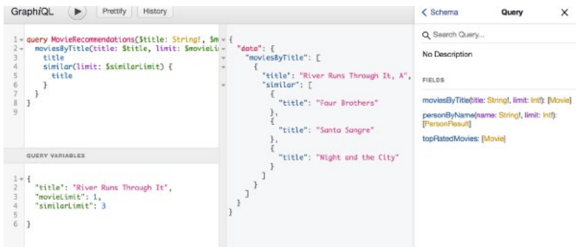
```
const schema = makeExecutableSchema({
  typeDefs,
  resolvers,
});
const app = express().use('*', cors());
app.use('/graphql', bodyParser.json(), graphqlExpress({
  schema,
  context: {},
}));
app.use('/graphiql', graphiqlExpress({
  endpointURL: '/graphql'
}));
app.listen(8080, () => console.log(
  `GraphQL Server running on http://localhost:8080/graphql`
));
```

QUERYING A GRAPHQL SERVICE

As most GraphQL services use HTTP as the client-server protocol, we can use any tools capable of sending an HTTP request to query a GraphQL endpoint. However, there are developer tools designed specifically for working with GraphQL endpoints that offer advantages.

GRAPHIQL

One of the easiest ways to query a GraphQL endpoint is to use GraphiQL. GraphiQL is an in-browser IDE for GraphQL that allows for viewing a GraphQL schema through introspection, defining variables, and autocompletion and validation for GraphQL queries. Tools such as graphql-express include options for serving GraphiQL alongside the GraphQL endpoint.



```
query MovieRecommendations($title: String!, $m
  moviesByTitle(title: $title, limit: $movieLi
    title
    similar(limit: $similarLimit) {
      title
    }
  }
}
```

```
{
  "title": "River Runs Through It",
  "movieLimit": 1,
  "similarLimit": 3

}
```

```
{
  "data": {
    "moviesByTitle": [
      {
        "title": "River Runs Through It, A",
        "similar": [
          {
            "title": "Four Brothers"
          },
          {
            "title": "Santa Sangre"
          },
          {
            "title": "Night and the City"
          }
        ]
      }
    ]
  }
}
```

No Description

FIELDS

moviesByTitle(title: String!, limit: Int!): [Movie]

personByName(name: String!, limit: Int!): [PersonResult]

topRatedMovies: [Movie]

APOLLO CLIENT

GraphQL clients such as Apollo Client or Relay simplify building GraphQL-based applications and offer benefits such as static queries, query parsing, data caching, and optimistic UI features. GraphQL clients, such as Apollo Client, offer integrations with frontend frameworks, such as React, to offer idiomatic approaches for querying GraphQL specific to a developer's framework of choice. Here is a generic example of using Apollo Client:

```
const client = new ApolloClient({
  networkInterface: createNetworkInterface({
    uri: 'http://localhost:8080/graphql/',
    opts:{
      headers: {
        Authorization: "Basic Z3JhcGhxbDpncmFwaHFs"
      }
    }
  })
})

import gql from 'graphql-tag';

client.query({
  query: gql`
  {
    moviesByTitle(title: "Matrix Reloaded", limit: 3) {
      title
      similar(limit: 2) {
        title
        year
      }
    }
  }
  `,
})
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

ADVANCED GRAPHQL

EXTENDING GRAPHQL WITH DIRECTIVES

The GraphQL specification includes a feature called a directive, which can be attached to any field or type (either in the schema or in a GraphQL query). The core GraphQL specification includes only two directives: `@include(if: Boolean)` and `@skip(if: Boolean)`. However, directives provide a mechanism for extending a GraphQL implementation while remaining fully compliant with the GraphQL specification.

For example:

- The `@relation` directive used by Graphcool to indicate relation fields (types that contain references to other types) in the GraphQL schema definition. From their docs:

```
type User {
 id: ID!
 stories: [Story!]! @relation(name: "UserOnStory")
}

type Story {
 id: ID!
 text: String!
 author: User! @relation(name: "UserOnStory")
}
```

- The neo4j-graphql integration allows for annotating a GraphQL field with a Cypher query using the `@cypher` directive, mapping the result of that Cypher query to the GraphQL field:

```
colleagues: [Person] @cypher(statement: "MATCH (this)-[:ACTED_
IN]->()<-[:ACTED_IN]-(other) RETURN other")
```

SUBSCRIPTIONS

Often, clients want to receive updates from the server when certain data changes. In addition to queries and mutations, GraphQL supports a third operation called a **subscription**, which allows a client to subscribe to receive event updates. Clients specify the data they want to subscribe to by sending a subscription query. Let's say we want to receive an update whenever a user submits a movie review for a particular movie. If the user is viewing a movie page, we want to update the UI when a new review is received:

```
subscription MovieReviewSubscription($movieId: ID!) {
  movieReviewSubscription(movieId: $movieId) {
    movie {
      title
    }
    rating
  }
}
```

You can learn more about using subscriptions with Apollo Client here.

This Refcard has introduced GraphQL concepts and demonstrated how we can build a GraphQL service using JavaScript. We've covered some of the benefits of GraphQL, especially the ability of the client to request specific data fields and reduce the number of roundtrip requests. However, there are still many topics left to explore. The next section provides an overview of further resources available for learning more about GraphQL and tools to simplify the process of building GraphQL applications.

The full code example used in this refcard is available as an Apollo Launchpad here.

THE GRAPHQL ECOSYSTEM

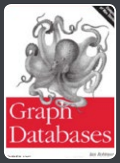
There are many great tools in the GraphQL ecosystem to help you learn more about GraphQL and build GraphQL applications. Here are a few:

GraphiQL	An in-browser IDE for exploring GraphQL
graphql-tools	An opinionated structure for how to build a GraphQL schema and resolvers in JavaScript, following the GraphQL-first development workflow
Apollo Client	A flexible GraphQL client for React and native apps
Apollo Launchpad	An in-browser GraphQL server playground
Graphcool	A serverless GraphQL backend platform
DataLoader	A utility for batching and caching requests, often used with GraphQL services
Join Monster	Translates GraphQL queries to SQL based on schema definition
PostGraphQL	Create a GraphQL API by reflection over a PostgreSQL schema
neo4j-graphql	GraphQL bindings for Neo4j graph database, generates and runs Cypher from GraphQL
Graphql.org	Portal for all things GraphQL
graphql-js	A GraphQL reference implementation for JavaScript
GraphQL specification	A working draft of the GraphQL specification, maintained by Facebook
Awesome GraphQL	A collection of links to GraphQL resources including tutorials, libraries, and examples

ABOUT THE AUTHOR

WILLIAM LYON is a software engineer on the Developer Relations team at Neo4j, the open source graph database, where he builds tools for integrating Neo4j with other technologies and helps developers be successful with graphs. Prior to Neo4j, he worked as a software engineer for a variety of startups, building APIs, quantitative trading tools, and mobile apps for iOS. William holds a master's degree in Computer Science from the University of Montana. You can find him online at lyonwj.com or @lyonwj

RECOMMENDED BOOK



FREE DOWNLOAD

Graphs model the real world beautifully. But in a graph database, the absence of a natural schema can make both design and implementation difficult. *Graph Databases* helps you solve graph modeling and querying problems by walking through every aspect of graph database development, from basics of graph theory to predictive analysis with graphs to graph database internals.