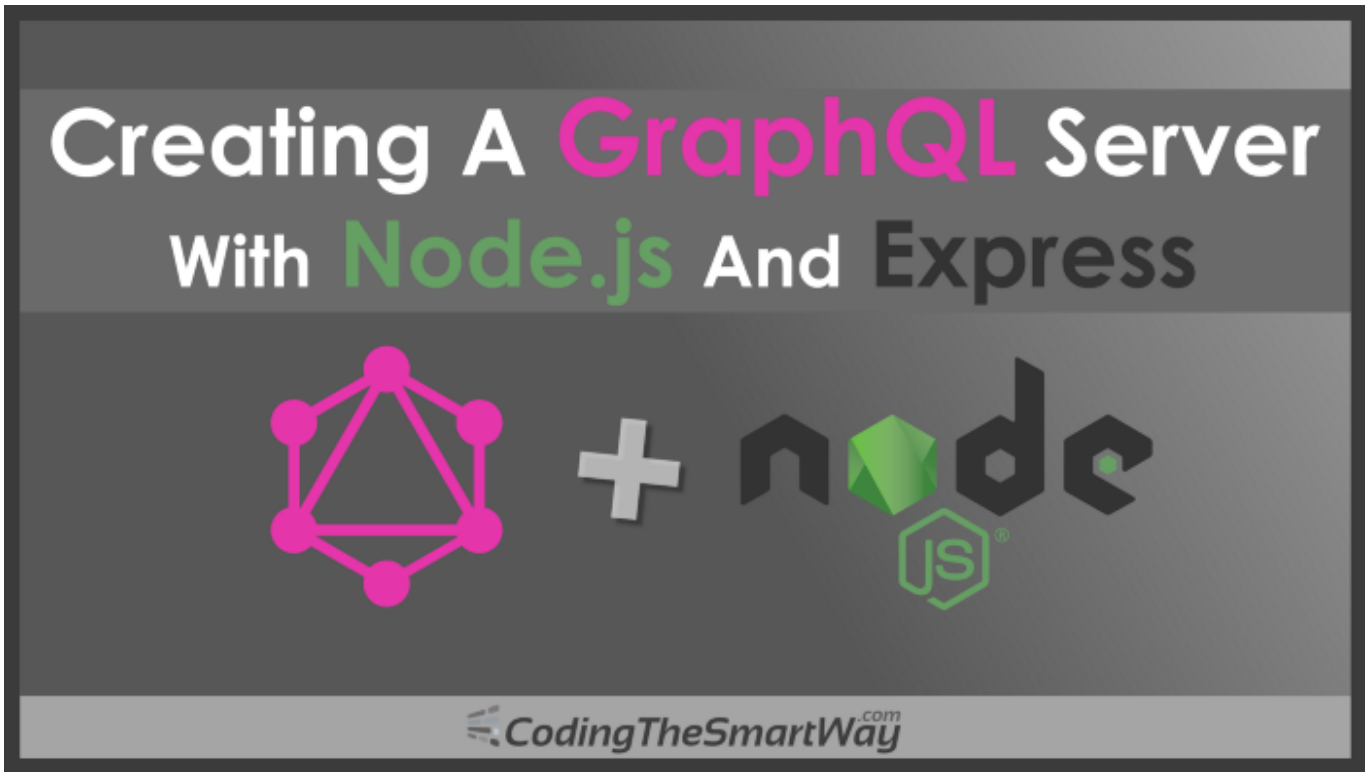
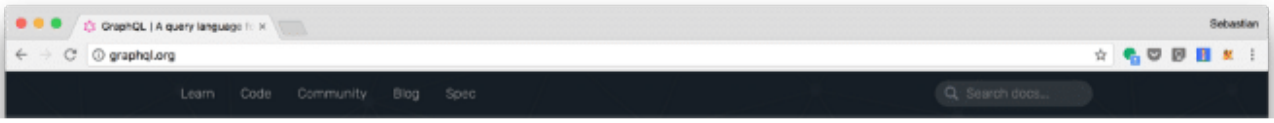


Creating A GraphQL Server With Node.js And Express

Sebastian Eschweiler Follow
Jan 21, 2018 · 10 min read

This post has been published first on CodingTheSmartWay.com.

GraphQL is a language that enables you to provide a complete and understandable description of the data in your API. Furthermore it gives clients the power to ask for exactly what they need and nothing more. The project's website can be found at http://graphql.org/.



```
var express = require('express');
var express_graphql = require('express-graphql');
var { buildSchema } = require('graphql');

// GraphQL schema
var schema = buildSchema(`
    type Query {
        message: String
    }
`);

// Root resolver
var root = {
    message: () => 'Hello World!'
};

// Create an express server and a GraphQL endpoint
var app = express();
app.use('/graphql', express_graphql({
    schema: schema,
    rootValue: root,
    graphiql: true
}));
app.listen(4000, () => console.log('Express GraphQL Server Now
Running On localhost:4000/graphql'));
```

At first we're making sure that *express*, *express-graphql* and the *buildSchema* function from the *graphql* package are imported. Next we're creating a simple GraphQL schema by using the *buildSchema* function.

To create the schema we're calling the function and passing in a string that contains the IDL (GraphQL Interface Definition Language) code which is used to describe the schema. A GraphQL schema is used to describe the complete APIs type system. It includes the complete set of data and defines how a client can access that data. Each time the client makes an API call, the call is validated against the schema. Only if the validation is successful the action is executed. Otherwise an error is returned.

Finally the Express server is created with a GraphQL endpoint: */graphql*. To create the GraphQL endpoint first a new express instance is stored in app. Next the *app.use* method is called and two parameters are provided:

- First the URL endpoint as string
- Second the result of the express_graphql function is handed over. A configuration object is passed into the call of express_graphql containing three properties

The three configuration properties which are used for the Express GraphQL middleware are the following:

- *schema*: The GraphQL schema which should be attached to the specific endpoint
- *rootValue*: The root resolver object
- *graphiql*: Must be set to *true* to enable the GraphiQL tool when accessing the endpoint in the browser. GraphiQL is a graphical interactive in-browser GraphQL IDE. By using this tool you can directly write your queries in the browser and try out the endpoint.

Finally *app.listen* is called to start the server process on port 4000.

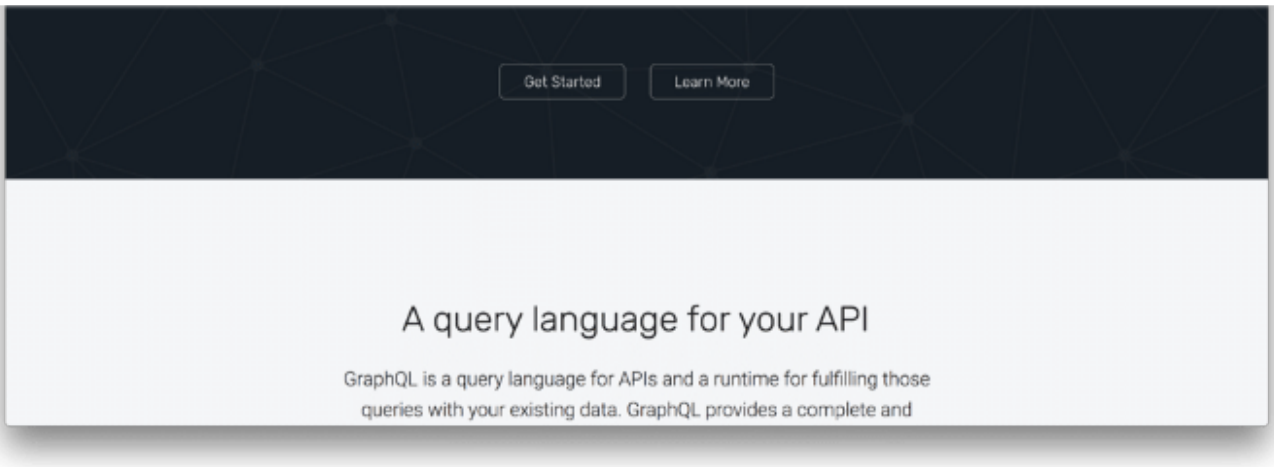
The Node.js server can be started by executing the following command in the project directory:

```
$ node server.js
```

Having started the server process you should be able to see the output

Express GraphQL Server Now Running On localhost:4000/graphql

on the command line. If you access *localhost:4000/graphql* in the browser you should be able to see the following result:



There are several advantages of GraphQL.

GraphQL is declarative: Query responses are decided by the client rather than the server. A GraphQL query returns exactly what a client asks for and no more.

GraphQL is compositional: A GraphQL query itself is a hierarchical set of fields. The query is shaped just like the data it returns. It is a natural way for product engineers to describe data requirements.

GraphQL is strongly-typed: A GraphQL query can be ensured to be valid within a GraphQL type system at development time allowing the server to make guarantees about the response. This makes it easier to build high-quality client tools

In this tutorial you'll learn how to setup a GraphQL server with Node.js and Express. We'll be using the Express middleware *express-graphql* in our example. Furthermore you'll learn how to use GraphQL on the client side to send queries and mutations to the server.

Let's get started …

Creating A GraphQL Server With Node.js And Expr...

Setting Up The Project

To setup a GraphQL Node.js server let's start with creating a new empty project folder first:

```
$ mkdir gql-server
```

Change into that directory and initiate a new *package.json* file by executing the following NPM command:

```
$ npm init
```

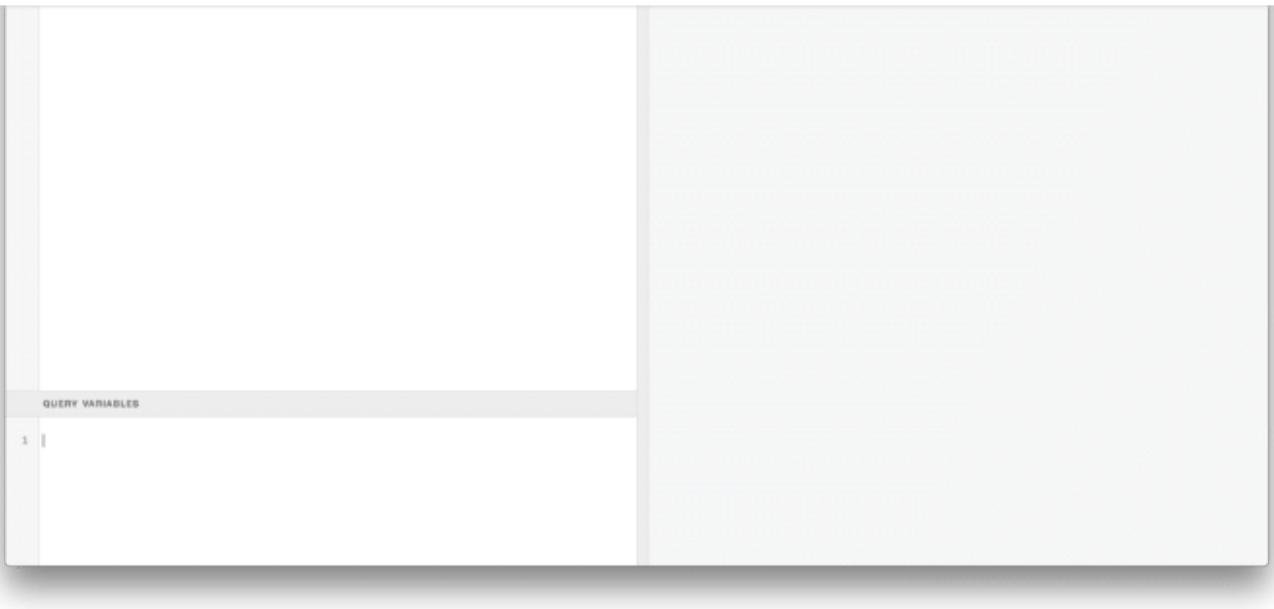
Furthermore create a new server.js file in the project directory. That will be the file where the code required to implement the Node.js GraphQL server will be inserted in the next section:

```
$ touch server.js
```

Finally make sure that NPM packages *graphql*, *express* and *express-graphql* are added to the project:

```
$ npm install graphql express express-graphql —save
```

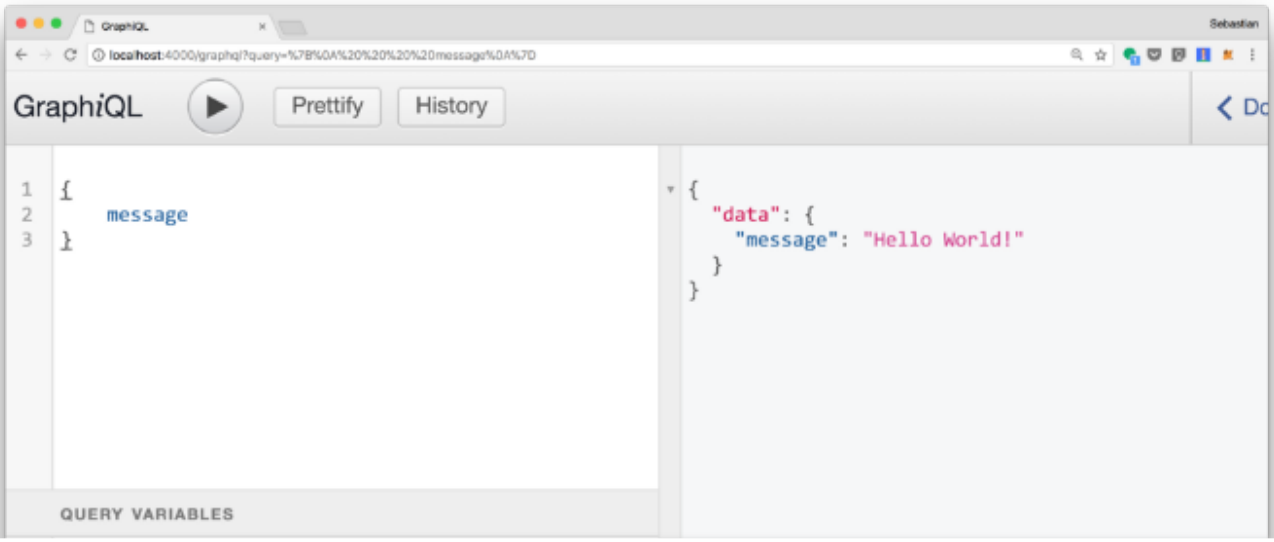
Having installed these packages successfully we're now ready to implement a first GraphQL server.



In the query editor type in the following code:

```
{
    message
}
```

Next hit the *Execute Query* button and you should be able to see the following result:



Implementing A More Sophisticated Example

Now that you have a basic understanding of how to implement a GraphQL server with Node.js and Express, let's continue with a more sophisticated example. Add a new JS file to the project:

```
$ touch server2.js
```

Next let's add the following implementation:

```
var express = require('express');
var express_graphql = require('express-graphql');
var { buildSchema } = require('graphql');

// GraphQL schema
var schema = buildSchema(`
    type Query {
        course(id: Int!): Course
        courses(topic: String): [Course]
    },
    type Course {
        id: Int
        title: String
        author: String
        description: String
        topic: String
        url: String
    }
`);

var coursesData = [
    {
        id: 1,
        title: 'The Complete Node.js Developer Course',
        author: 'Andrew Mead, Rob Percival',
        description: 'Learn Node.js by building real-world
applications with Node, Express, MongoDB, Mocha, and more!',
        topic: 'Node.js',
```

```
        description: 'Learn by example building & deploying real
world Node.js applications from absolute scratch',
        topic: 'Node.js',
        url: 'https://codingthesmartway.com/courses/nodejs-express-
mongodb/'
    },
    {
        id: 3,
        title: 'JavaScript: Understanding The Weird Parts',
        author: 'Anthony Alicea',
        description: 'An advanced JavaScript course for everyone!
Scope, closures, prototypes, this, build your own framework, and
more.',
        topic: 'JavaScript',
        url: 'https://codingthesmartway.com/courses/understand-
javascript/'
    }
]

var getCourse = function(args) {
    var id = args.id;
    return coursesData.filter(course => {
        return course.id == id;
    })[0];
}

var getCourses = function(args) {
    if (args.topic) {
        var topic = args.topic;
        return coursesData.filter(course => course.topic === topic);
    } else {
        return coursesData;
    }
}

var root = {
    course: getCourse,
    courses: getCourses
};

// Create an express server and a GraphQL endpoint
var app = express();
app.use('/graphql', express_graphql({
    schema: schema,
    rootValue: root,
    graphiql: true
}));
```

of a custom type *Course* and two query actions.

The Course object type consist of six properties in total. The defined query actions enable the user to retrieve a single course by ID or retrieving an array of Course objects by course topic.

To be able to return data without the need to connect to a database we're defining the *coursesData* array with some dummy course data inside.

In the root resolver we're connecting the *course* query action to the *getCourse* function and the *courses* query action to the *getCourses* function.

Accessing The GraphQL API

Now let's start the Node.js server process again and execute the code from file *server2.js* with the following command:

```
$ node server2.js
```

If you're opening up URL *localhost:4000/graphql* in the browser you should be able to see the GraphiQL web interface, so that you can start typing in queries. First let's retrieve one single course from our GraphQL endpoint. Insert the following query code:

```
query getSingleCourse($courseID: Int!) {
    course(id: $courseID) {
        title
        author
        description
        topic
        url
    }
}
```

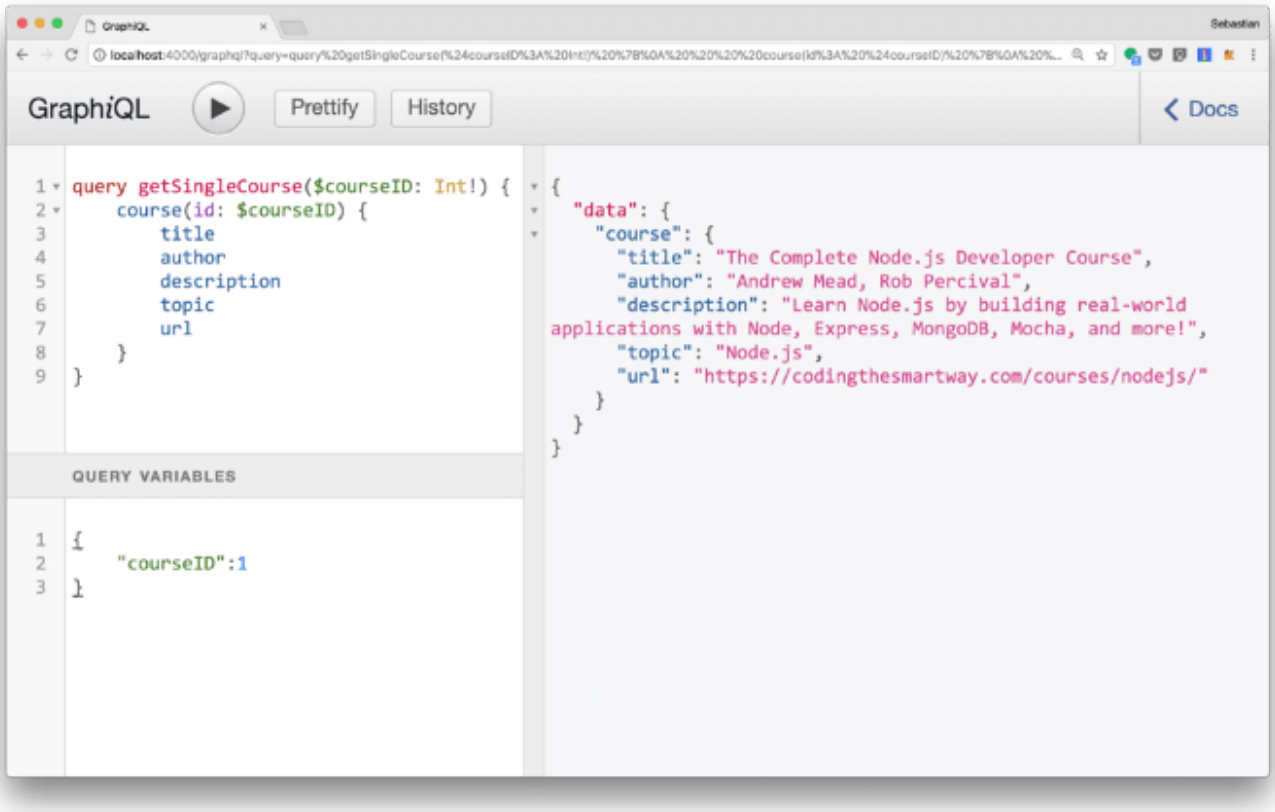
The *getSingleCourse* query operation is expecting to get one parameter: *$courseID* of type *Int*. By usign the exclamation mark we're specifying that this parameters needs to be

that specific course.

Because the *getSingleCourse* query operation uses a dynamic parameter we need to supply the value of this parameter in the Query Variables input field as well:

```
{
    "courseID":1
}
```

Click on the execute button and you should be able to see the following result:



Using Aliases & Fragments

You're able to include multiple queries in one query operation. In the following example the getCourseWithFragments query operations contains two queries for single courses.

```
            ...courseFields
      },
      course2: course(id: $courseID2) {
            ...courseFields
      }
}

fragment courseFields on Course {
  title
  author
  description
  topic
  url
}
```

As you can see the query operations requires two parameters: courseID1 and courseID2. The first ID is used for the first query and the second ID is used for the second query.

Another feature which is used is a fragment. By using a fragment we're able to avoid repeating the same set of fields in multiple queries. Instead we're defining a reusable fragment with name courseFields and specific which fields are relevent for both queries in one place.

Before executing the query operation we need to assign values to the parameters:

```
{
    "courseID1":1,
    "courseID2":2
}
```

The result should look like the following:

Creating And Using Mutations

So far we've only seen examples which fetches data from our GraphQL server. With GraphQL we're also able to modify data. by using Mutations. To be able to use a mutation with out GraphQL server we first need to add code to our server implementation in *server2.js*:

```
// GraphQL schema
var schema = buildSchema(`
    type Query {
        course(id: Int!): Course
        courses(topic: String): [Course]
    },
    type Mutation {
        updateCourseTopic(id: Int!, topic: String!): Course
    }
    type Course {
        id: Int
        title: String
        author: String
        description: String
        topic: String
        url: String
    }
`);
```

Using that mutation it should be possible to change the topic of a specific course. In the same way like we did it before for queries we're now assigning a function to the mutation in the root resolver. The function is implemented with the corresponding update logic:

```
var updateCourseTopic = function({id, topic}) {
    coursesData.map(course => {
        if (course.id === id) {
            course.topic = topic;
            return course;
        }
    });
    return coursesData.filter(course => course.id === id) [0];
}

var root = {
    course: getCourse,
    courses: getCourses,
    updateCourseTopic: updateCourseTopic
};
```

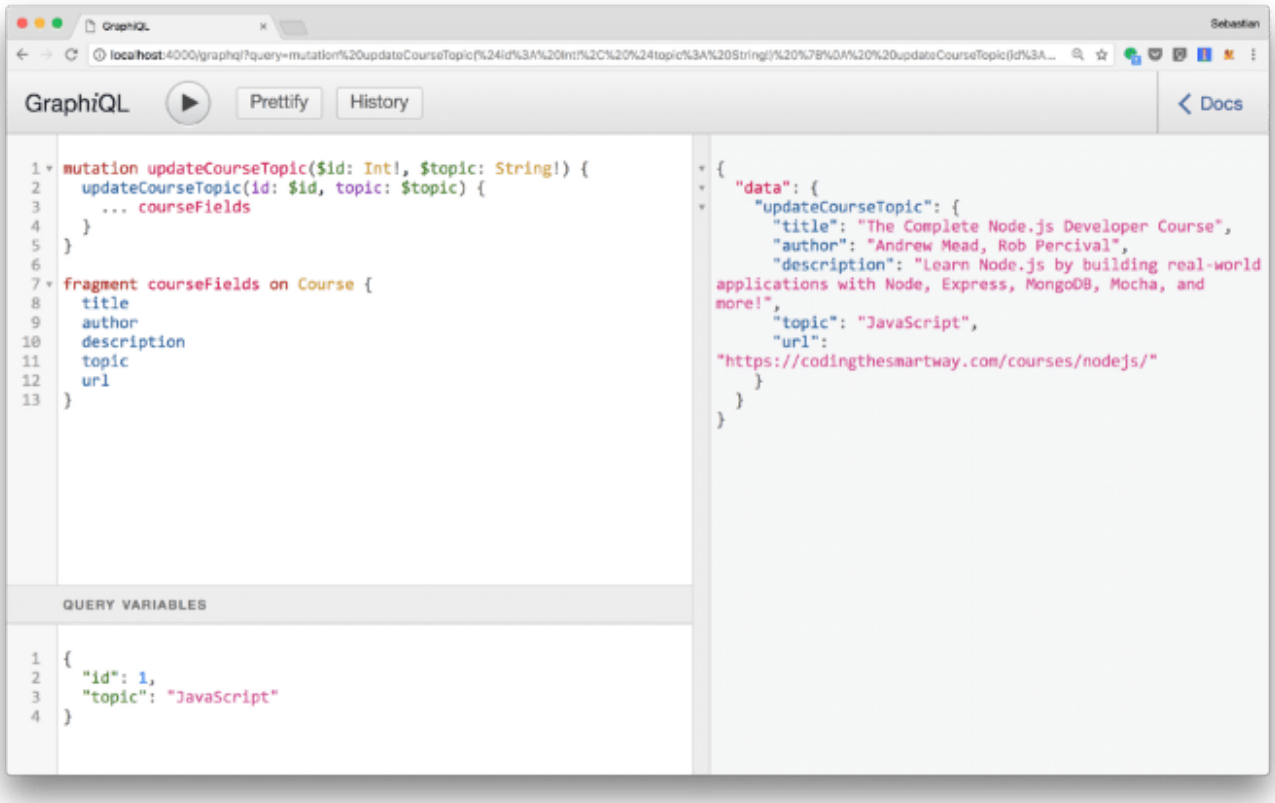
Now the sever is able to handle mutations as well, so let's try it out in the GraphiQL browser interface again.

A mutation operation is defined by using the mutation keyword followed by the name of the mutation operation. In the following example the *updateCourseTopic* mutation is included in the operation and again we're making use of the *courseFields* fragment.

```
mutation updateCourseTopic($id: Int!, $topic: String!) {
  updateCourseTopic(id: $id, topic: $topic) {
    ... courseFields
  }
}
```

```
  "id": 1,
  "topic": "JavaScript"
}
```

By executing this mutation we're changing the value of the *topic* property for the course data set with ID 1 from *Node.js* to *JavaScript*. As a result we're getting back the changed course:



Conclusion

GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more.

In this tutorial you've learned how to implement your own GraphQL server with Node.js and Express. By using the Express middleware *express-graphql* setting up a GraphQL

. . .

Top Online Course Recommendations

#1 GraphQL with React: The Complete Developers Guide

Learn and master GraphQL by building real web apps with React and Node
Go To Course …

#2 The Complete Web Developer in 2018

Learn to code and become a web developer in 2018 with HTML5, CSS, Javascript, React, Node.js, Machine Learning & more!
Go To Course …

#3 The Complete JavaScript Course — Build a Real-World Project

Master JavaScript with the most complete JavaScript course on the market! Includes projects, challenges, final exam, ES6
Go To Course …

Disclaimer: This post contains affiliate links, which means that if you click on one of the product links, I'll receive a small commission. This helps support this blog!

GraphQL React Reactjs Web Development JavaScript