How To Set Up a Powerful API With GraphQL, Koa, and MongoDB — CRUD

Indrek Lasn
Jan 30, 2019 · 6 min read ★

This is a series where we learn how to set up a powerful API with GraphQL, Koa, and Mongo. The primary focus will be on GraphQL. Check out part I of this article series, if you haven't yet.

Mutations

So far we can read our data, but there's a good chance we need to edit our data records/documents. Any complete data platform needs a way to modify server-side data as well.

Imagine that a company has launched a new gadget. How would we go about adding the record to our database with GraphQL?

What Are Mutations?

Think of mutations like `POST` or `PUT` REST actions. Setting up a mutation is quite straightforward.

Let's jump in!

Adding Records to Our Database

Create a file `graphql/mutations.js`.

Inside the file, we will place mutations.

```
const { GraphQLObjectType, GraphQLObjectType } = require('graphql');
const gadgetGraphQLType =  require('./gadgetType');
const Gadget = require('./../models/gadget');

const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {


  }
})

module.exports = Mutation;
```

mutations is hosted with ♡ by GitHub view raw

- We'll import the `GraphQLObjectType` and `GraphQLObjectType` objects from the GraphQL library.
- Import the GraphQL type for `gadget` .
- Import the `gadget` Mongoose model.

After importing the things we need, we can create the mutation.

A mutation is just a plain `GraphQLObjectType` , like the query we had before. It has two main properties we're interested in.

1. The name of the mutation is what appears in the `graphiql` docs.

2. Fields are where we can place our mutation logic.

```
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addGadget: {
      // add props here
    }
  }
})
```

mutations.js hosted with ♡ by GitHub view raw

Notice I added a new object inside the `fields` object. It's called `addGadget` , and it will do exactly what it says it'll do.

Inside the `addGadget` we have access to three properties, `type` , `args` , and `resolve()` .

```
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addGadget: {
      type: gadgetGraphQLType,
    }
  }
})
```

mutations.js hosted with ♡ by GitHub view raw

The `addGadget` type will be `gadgetGraphQLType`. The gadget can only have properties that are allowed in the `gadgetGraphQLType` type we declared earlier.

`addGadget` is a query that accepts arguments. The arguments are needed to specify which gadget we want to add to our database.

```
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addGadget: {
      type: gadgetGraphQLType,
      args: {
        name: { type: GraphQLString },
        release_date: { type: GraphQLString },
        by_company: { type: GraphQLString },
        price: { type: GraphQLString }
      },
    }
  }
})
```

mutations.js hosted with ♡ by GitHub view raw

We declare up front which arguments the query accepts, and the types of the arguments.

Lastly — what happens with the query? This is precisely why we have the `resolve()` function.

Remember the `resolve()` function has two arguments: `parent` and `args`. We're interested in the `args`, since these are the values we pass to our query.

```
resolve(parent, args) {
  // Create a new mongo record
}
```

mutations.js hosted with ♡ by GitHub view raw

Inside the resolve, we place the logic for creating a new Mongo record.

```
const newGadget = new Gadget({
  name: <a href="http://args.name" class="link link-url" target="_blank" rel="external nofollow
  release_date: args.release_date,
```

```
    by_company: args.by_company,
    price: args.price,
  })

  return newGadget.save();
```

mutations.js hosted with ♡ by GitHub view raw

We create a new instance of our `Gadget` Mongoose model, pass the props we receive from GraphQL as new fields and finally save the record.

Here's how the full mutation looks:

`graphl/mutations.js`

```
const { GraphQLObjectType, GraphQLString } = require('graphql');
const gadgetGraphQLType =  require('./gadgetType');
const Gadget = require('./../models/gadget');

const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addGadget: {
      type: gadgetGraphQLType,
      args: {
        name: { type: GraphQLString },
        release_date: { type: GraphQLString },
        by_company: { type: GraphQLString },
        price: { type: GraphQLString }
      },
      resolve(parent, args) {
        const newGadget = new Gadget({
          name: <a href="http://args.name" class="link link-url" target="_blank" rel="external
          release_date: args.release_date,
          by_company: args.by_company,
          price: args.price,
        })

        return newGadget.save();
      }
    }

  }
})

```

```
31  module.exports = Mutation;
```

mutations.js hosted with ♡ by GitHub view raw

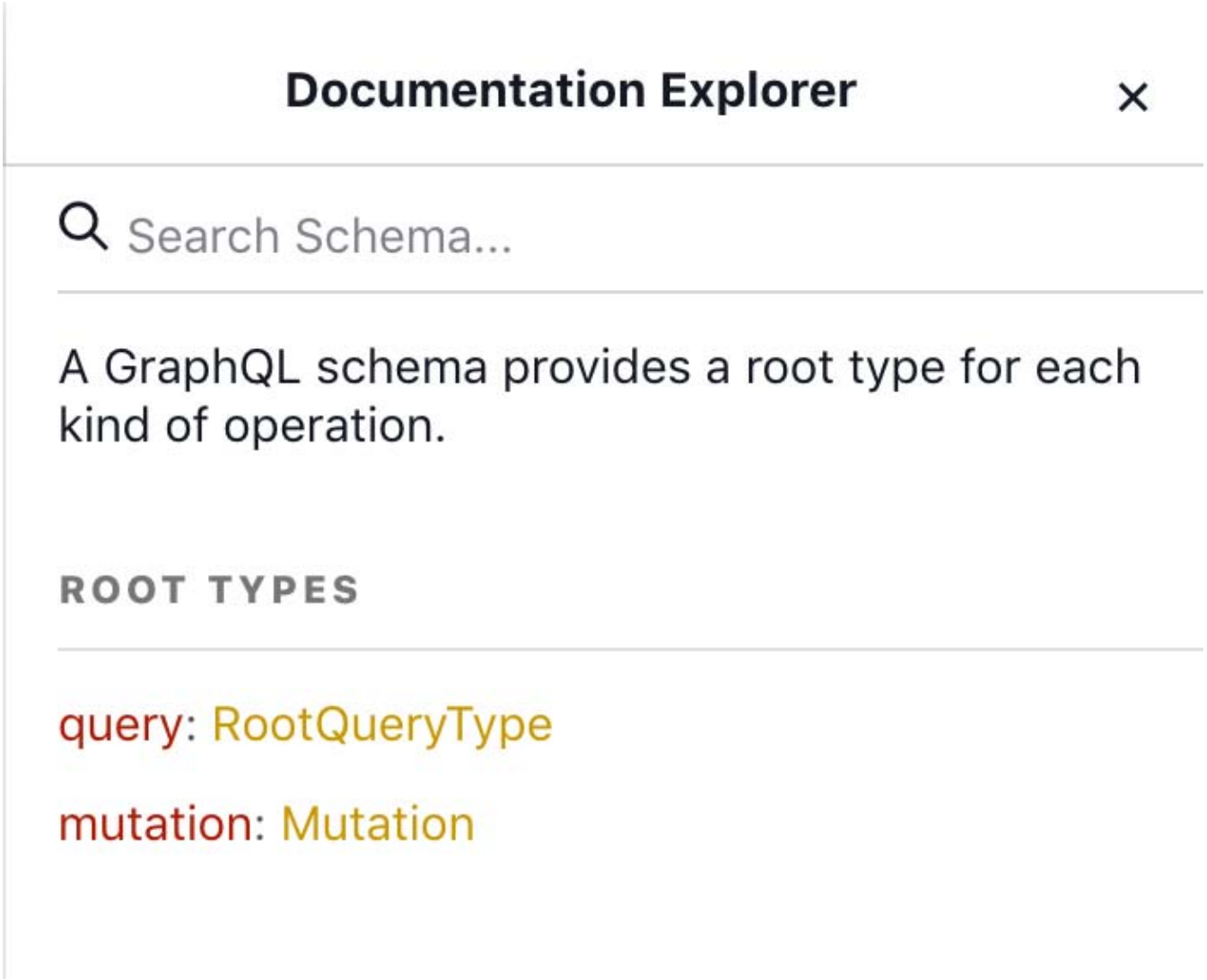
Voilà! All we need to do is import the mutation to our `schema.js` file.

`graphl/schema.js`

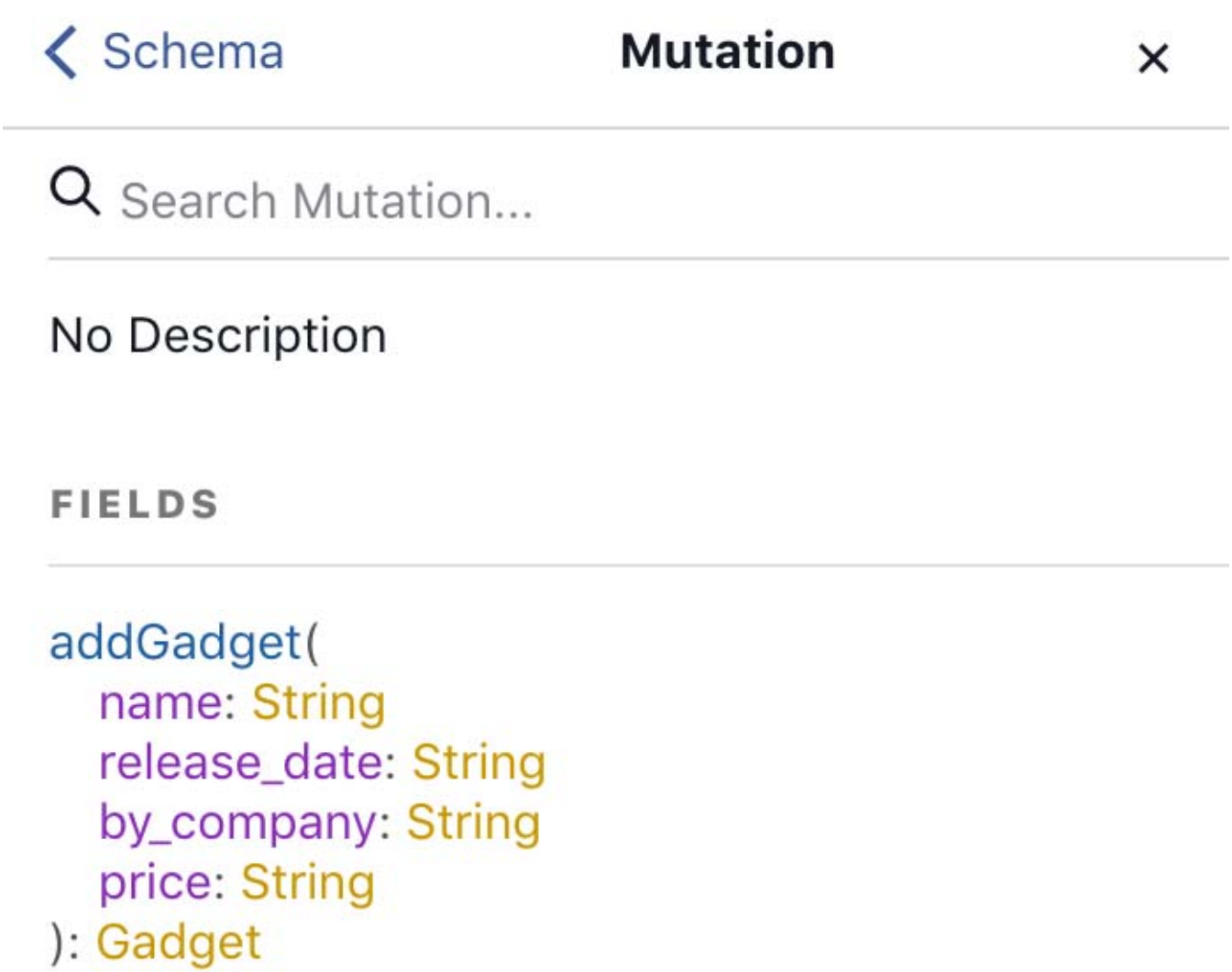
```
1  const Mutations = require('./mutations');
2
3  /* stuff */
4
5  module.exports = new GraphQLSchema({
6    query: RootQuery,
7    mutation: Mutations
8  });
```

schema.js hosted with ♡ by GitHub view raw

If everything went fine, this is what we should see on our GraphiQL:



And if we click on it:



Notice how GraphQL automatically creates self-documentation.

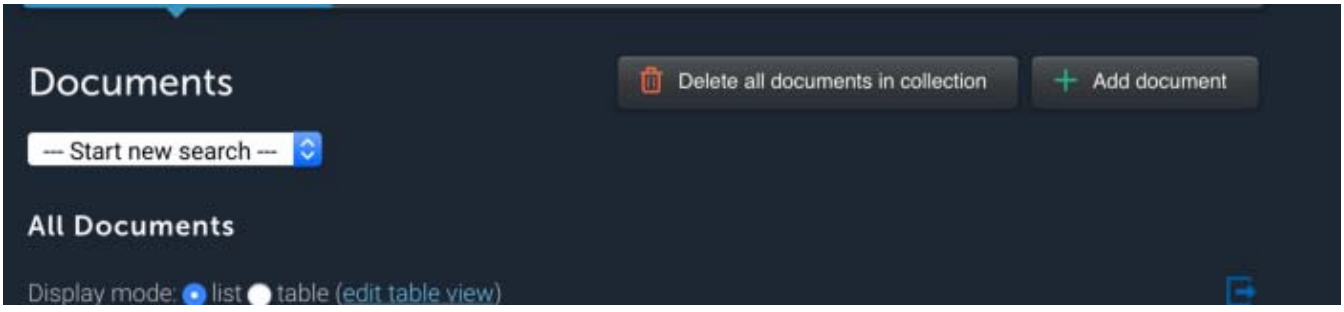
Firing Off the Mutation Query

A mutation is just a plain GraphQL query, which takes our arguments, saves it to the Mongo database, and returns the properties we want.

Here's the catch — every mutation needs to be marked as `mutation`:

We've successfully created and inserted a new gadget to our Mongo database.

If you head over to mLab, or whatever provider you're using, you should see the new record.



Here's the complete query for our mutation.

```
mutation {
  addGadget(name: "MacBook Pro", release_date: "January 10, 2006", by_company: "Apple", price: '
  name,
  release_date,
  by_company,
  price
  }
}
```

addGadget.gql hosted with ♡ by GitHub view raw

Editing Our Records in the Database

What if we want to edit pre-existing records? We can't rely on never making a mistake, or what if the price changes?

Editing a record is also a mutation. Remember, every time we want to change/add a new record, it's a GraphQL mutation!

Open the `graphql/mutations` file and create another mutation. A mutation is just a plain object.

```
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
```

```
  fields: {
    addGadget: {
      type: gadgetGraphQLType,
      args: {
        name: { type: GraphQLString },
        release_date: { type: GraphQLString },
        by_company: { type: GraphQLString },
        price: { type: GraphQLString }
      },
      resolve(parent, args) {
        const newGadget = new Gadget({
          name: <a href="http://args.name" class="link link-url" target="_blank" rel="external
          release_date: args.release_date,
          by_company: args.by_company,
          price: args.price,
        })

        return newGadget.save();
      }
    }, // add new mutation
    updateGadget: {
      type: gadgetGraphQLType,
      args: {
        id: { type: GraphQLString },
        name: { type: GraphQLString },
        release_date: { type: GraphQLString },
        by_company: { type: GraphQLString },
        price: { type: GraphQLString }
      },
      resolve(parent, args) {

      }
    }
  }
})
```

mutations.is hosted with ♡ by GitHub view raw

Notice the new mutation is called `updateGadget`. It's pretty much a replica of the previous mutation. Notice the extra argument, the `id` — that's because we need to find the existing gadget and change it. We can find the gadget by id.

The `resolve()` function is where it gets more interesting. Ideally, we want to find the gadget by id, change the props, and save it. How would we go about doing this?

Mongoose gives us a method to do this, called `findById`.

```
resolve(parent, args) {
  return Gadget.findById(<a href="http://args.id" class="link link-url" target="_blank" rel="ext
}
```

mutations.js hosted with ♡ by GitHub view raw

This returns a promise. If we `console.log` the promise, we can see a huge blob of properties attached to it. What we can do with the promise, is chain it with a `then()` method.

```
resolve(parent, args) {
  return Gadget.findById(<a href="http://args.id" class="link link-url" target="_blank" rel="ex
    .then(gadget => {
      <a href="http://gadget.name" class="link link-url" target="_blank" rel="external nofollow
      gadget.release_date = args.release_date,
      gadget.by_company = args.by_company,
      gadget.price = args.price

      return gadget.save()
    })
}
```

mutations.js hosted with ♡ by GitHub view raw

So, we find the gadget, change the props, and save it. But this returns another promise that we need to resolve.

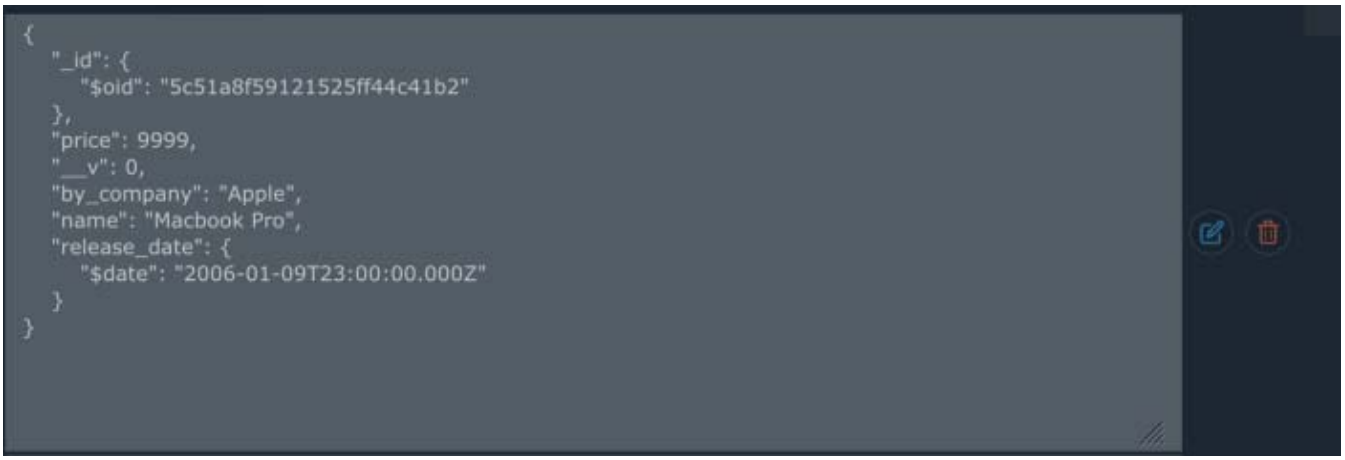
```
resolve(parent, args) {
  return Gadget.findById(<a href="http://args.id" class="link link-url" target="_blank" rel="ex
    .then(gadget => {
      <a href="http://gadget.name" class="link link-url" target="_blank" rel="external nofollow
      gadget.release_date = args.release_date,
      gadget.by_company = args.by_company,
      gadget.price = args.price
      return gadget.save()
    })
    .then(updatedGadget => updatedGadget)
    .catch(err => console.log(err))
}
```

mutations.js hosted with ♡ by GitHub view raw

`.catch()` for error handling, in case we run into errors. Remember, you can monitor your `pm2` logs via the `pm2 logs` command. If you run into errors, these will be logged to the pm2 logger.

That's all! Query time. Look at your Mongo table and pick a random `id` from there, then edit the corresponding gadget.

And if we inspect the database, we should see the edited record.



Success!

Here's the query for the `updateGadget` mutation.

```
mutation {
  updateGadget(
    id: "5c51a8f59121525ff44c41b2", name: "Macbook Pro",
    release_date: "January 10, 2006", by_company: "Apple",
    price: "9999") {
```

```
      price: 9999 ) {
      id
      name
      release_date
      by_company
      price
    }
  }
```

updateGadget.gql hosted with ♡ by GitHub view raw

Okay, so far we have the `Create`, `Read`, and `Update`, but we're missing the final `d` (elete).

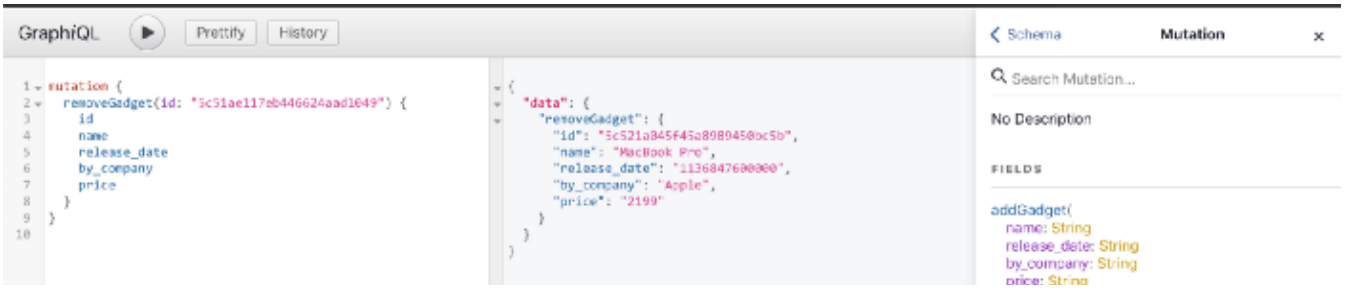
Deleting a record from a Mongo database is quite straightforward. All we need is another mutation, since we are, in fact, mutating the database.

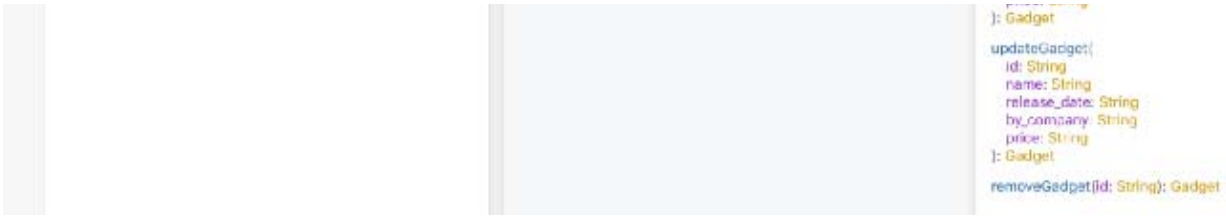
For deleting records, Mongoose gives us a handy method called `findOneAndDelete` — you can read more about findOneAndDelete here.

```
removeGadget: {
  type: gadgetGraphQLType,
  args: {
    id: { type: GraphQLString }
  },
  resolve(parent, args) {
    return Gadget.findOneAndDelete(<a href="http://args.id).exec(" class="link link-url" target
      .then(gadget => gadget.remove())
      .then(deletedGadget => deletedGadget)
      .catch(err => console.log(err))
  }
}
```

mutations.js hosted with ♡ by GitHub view raw

Deleting the record just takes one argument — the id. We find the gadget by id, delete it, and return it. If there's an error, we'll log it.





And the query:

```
mutation {
  removeGadget(id: "5c51ae117eb446624aad1049") {
    id
    name
    release_date
    by_company
    price
  }
}
```

removeQuery.js hosted with ♡ by GitHub view raw

Note: Make sure the `id` is correct and exists in the database, otherwise, it won't work.

If we head over to our database and inspect it — indeed the record got deleted from our database.

Well done, we have achieved basic `CRUD` functionality. Notice how GraphQL is a thin layer between our database and view. It's not supposed to replace a database, but rather make it easier to work with data, fetching, and manipulating.

If you're feeling good for GraphQL, I recommend reading through the "*The Road to GraphQL*" book for a more in-depth dive.

Here's the source code:

wesharehoodies/koa-graphql-mongodb

Tutorial how to set up koa with graphql and mongodb - wesharehoodies/koa-graphql-mongodb

github.com

Don't miss part three where we'll do more great stuff:

Thanks for reading!

GraphQL JavaScript API Nodejs Programming