Monday, January 21, 2019 GraphQL

Image management via GraphQL

> This post is 4 years old. (Or older!) Code samples may not work, screenshots may be missing and links could be broken. Although some of the content may be relevant please take it with a pinch of salt.

In this article, we'll review how to manage images using GraphQL by discussing an application that utilises GraphQL queries and mutations to display a user's photo.

> *If you're new to GraphQL, please check out this **introductory article**.*

> *Would you like to learn more about GraphQL? Check out the **Practical Guide to GraphQL - Become a GraphQL Ninja** course.*

GraphQL vs REST API

GraphQL is a query language for APIs, which means that we can be very specific about the data returned to the application. This is entirely

different from how REST APIs work: REST APIs specify a number of endpoints and those endpoints return some data structure. It is then up to the client to pick which parts of the data structure they want to use.

With GraphQL we can directly ask for the data the application needs which makes things more declarative.

> *Please note that REST and GraphQL do not compete with each other, they complement each other (for example, GraphQL can serve as a wrapper around an existing REST API). There are valid use cases for both and, they can be used together with ease within the same project.*

Queries and Mutation

As mentioned earlier GraphQL is primarily a query language concerned with fetching data. However, there are situations when we need to send data via a form. In this case, we can leverage what GraphQL calls mutations. Simply put with mutations we can create data, and if the mutation returns an object, we can tell GraphQL the specific fields to return. This also allows us to check the new state of the object once it has been updated.

> *Note that a mutation can also return a Boolean value, in which case there won't be any fields to query, but the mutation will return a true/false value.*

A typical mutation looks like this:

```
makeUser(id: Int!, name: String!): User!
```

This creates a user via a mutation called 'makeUser'. It requires an `**id**` and a `**name**` as parameters (both are required). It returns a custom type `**User**` (We need to define how the User type looks like as well)

```
type User {
  id: ID!
  name: String!
  photo: String
}
```

The photo field returns a `**String**` which is not a required field, hence it doesn't make sense to add it as a parameter to the mutation. However, it makes a lot of sense to have another mutation to handle the image upload. Remember, mutations are for creating data; therefore it is perfectly valid to use a mutation to do a file upload as well:

```
uploadImage(filename: String!, id: Int!): String!
```

What are those two parameters? When uploading a file, we need to specify a `**filename**` as well as an `**id**`. The `**id**` is going to denote the identity of the user having the image as a profile photo.

User model

Let's take a step back and see what the user model looks like:

```
{
  id: 1,
```

```
    name: 'Tamas',
    cars: [1, 2],
    photo: null
  }, { // ...
```

Users have `id`, `name`, `cars` and `photo` properties. Initially, the `photo` property is set to `null`.

> *Note that the `cars` property above references another model. (left out of the discussion for brevity)*

Express and Pug

To display data from the GraphQL instance, we'll create a basic Express application and use Pug for templating. This basic setup will focus on the bits that are relevant for this article:

```
// code snippet - app.js
const routes = require('./routes');
app.get('/user/:id', routes.userinfo);

// code snippet - routes.js
const userinfo = async (req, res) => {
  const { id } = req.params;
  if (id) {
    const query = `{
      user(id: ${id}) {
        id
        name
        photo
        car {
```

```
            id
            make
            model
            colour
          }
        }
      }`;
      const response = await fetchGraphQL(query);
      return res.render('user', {
        data: response.data.user,
      });
    }
    return res.status(400).send('Please provide an ID');
  };
```

The Pug template looks like this:

```
h3= data.name
if !data.photo
  p
    img(src='https://res.cloudinary.com/tamas-demo/image/upload
if data.photo
  p
    img(src=data.photo)
if data.car.length === 0
  p This user has no cars.
if data.car
  ul
    for car in data.car
      li #{car.make} #{car.model} (#{car.colour})

p Upload a profile photo:
form(method='post', action='/upload', enctype='multipart/form-d
  input(type='file', name='file')
```

```
    input(type='hidden', value=data.id name='id')
    input(type='submit', value='Upload')

  p
    a(href='/') Go back
```

There are two things that we should note from the above template. First, if the `photo` property is not set for a given user, a placeholder avatar image is loaded from Cloudinary. Second, there's a form to upload files. We'll see how the `/upload` endpoint works but before we get to that, let's briefly discuss image management.

Image management

For this example, we are going to be using Cloudinary - a cloud-based media management service. This means that we don't need to manage our media assets locally - Cloudinary will take care of this for us. It also has excellent capabilities to serve images, apply optimisations and transformations on pictures as well as videos.

The process that we'll implement will have 2 phases when it comes to the file upload:

1. Upload an image from a frontend to Node.js
2. From Node.js "forward" the image to Cloudinary for final storage

Later on, we'll also take a look at how to retrieve the image using a custom GraphQL scalar.

Uploading an image

In Node.js, multiple npm packages are concerned with file upload. We'll be using multer, and after installing it via `npm i multer` configure it for an Express application like this:

```
const multer = require('multer');
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/');
  },
  filename: function (req, file, cb) {
    cb(null, file.originalname);
  },
});
const upload = multer({ storage });
```

> *Please note that Apollo Server also has a file uploader*

The above code makes sure that the files sent via the frontend are stored in the `uploads` folder and it preserves the original filename as well (failing to add the second function will result in having a random filename).

`Multer` acts as a middleware, therefore we need to add it to a route definition:

```
app.post(
  '/upload',
  upload.fields([{ name: 'file' }, { name: 'id' }]),
```

```
    routes.upload
);
```

Notice the two fields in the code above. If we take a look at the upload form, we'll see that it sends both a file and the `id` of the current user. This is required because the mutation requires this `ID` as well.

> *In a proper application the file upload functionality would likely be used only when a user is logged in. In that case, there wouldn't be a need to have the hidden field because by definition we'd know the identity of the user that's currently logged in. The approach used in this article is for demonstration and education purposes.*

File upload and mutation

In addition to uploading the file to the cloud, we also need to mutate it by updating the user object with the final Cloudinary URL of the profile image.

To achieve this, we'll write code that handles the mutation and the file upload at the same time:

```
// execute GraphQL queries and mutations
async function fetchGraphQL(query, variables = {}) {
  const response = await fetch('http://localhost:3000/gra
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      query,
      variables,
```

```
    }),
  });
  return response.json();
}

// handle the file upload, as an Express route
const upload = async (req, res) => {
  const id = +req.body.id;
  const uploadedFile = req.files.file[0];
  const filename = uploadedFile.filename;
  const mutation = `
  mutation($filename: String!, $id: Int!) {
    uploadImage(filename: $filename, id: $id)
  }
  `;
  await fetchGraphQL(mutation, { filename, id });
  return res.redirect(req.get('referer'));
};
```

The first function accepts a query as its first parameter (which can be a GraphQL query or a mutation, and optionally a variables parameter) and sends the request via HTTP POST to the GraphQL processor.

The second function - **`upload`** - is the Express router handler that performs the file upload. It then calls the mutation and refreshes the page via **`res.redirect(req.get('referer'));`**.

The mutation uses **`variables`** which is used in GraphQL to handle dynamic values for a query. This method is preferred to avoid passing dynamic arguments directly to the query string (otherwise the client would have to manipulate the query string at runtime dynamically).

The mutation code works as follows:

1. Create a mutation that accepts two variables denoted by `` `$filename` `` and `` `$id` `` - each with their datatype.
2. Pass the variables' values to the actual mutation via their references (`` `$filename` `` and `` `$id` ``) and assign them to the right field (e.g. `` `filename: $filename` ``).

> *Note that variables can not only be used for mutations but also for queries. For example, consider a cookbook with a list of recipes. Querying for* `` `onlyVegetarian(vegetarian: $vegetarian) { recipe(vegetarian: $vegetarian} { prepTime, ingredients } }` `` *would allow us to "flip the switch" via a Boolean value to show only vegetarian dishes.*

The mutation itself

We have functionality in place to upload an image and store it locally, but we have to write the code for the mutation as well, which is going to update the user object as well as upload the image to Cloudinary.

> *Note we could have uploaded the image directly to Cloudinary, but for education purposes, we are instead showing it step-by-step.*

Let's install the Cloudinary Node.js SDK (`` `npm i cloudinary` ``) and configure it:

```
const cloudinary = require('cloudinary');
cloudinary.config({
  cloud_name: process.env.CLOUD_NAME,
  api_key: process.env.API_KEY,
```

```
  api_secret: process.env.API_SECRET,
});
```

Next, let's takeNext, let's take a look at the actual mutation:

```
uploadImage: async (parent, { id, filename }, { models }
  const path = require('path');
  const mainDir = path.dirname(require.main.filename);
  filename = `${mainDir}/uploads/${filename}`;

  try {
    const photo = await cloudinary.v2.uploader.upload(fi
      use_filename: true,
      unique: false,
    });

    const user = models.users[id - 1];
    user.photo = `${photo.public_id}.${photo.format}`;
    return `${photo.public_id}.${photo.format}`;
  } catch (error) {
    throw new Error(error);
  }
};
```

The code above grabs the file stored by the frontend to the **`uploads`** folder and uploads that to Cloudinary. It then updates the user object by adding the filename and the extension to the photo object. Originally, the user object looked like:

```
{
  id: 1,
  name: 'Tamas',
```

```
    cars: [1, 2],
    photo: null
  }, { // ...
```

And after running the mutation, it looks like:

```
{
  id: 1,
  name: 'Tamas',
  cars: [1, 2],
  photo: 'some-photo.jpg'
}, { // ...
```

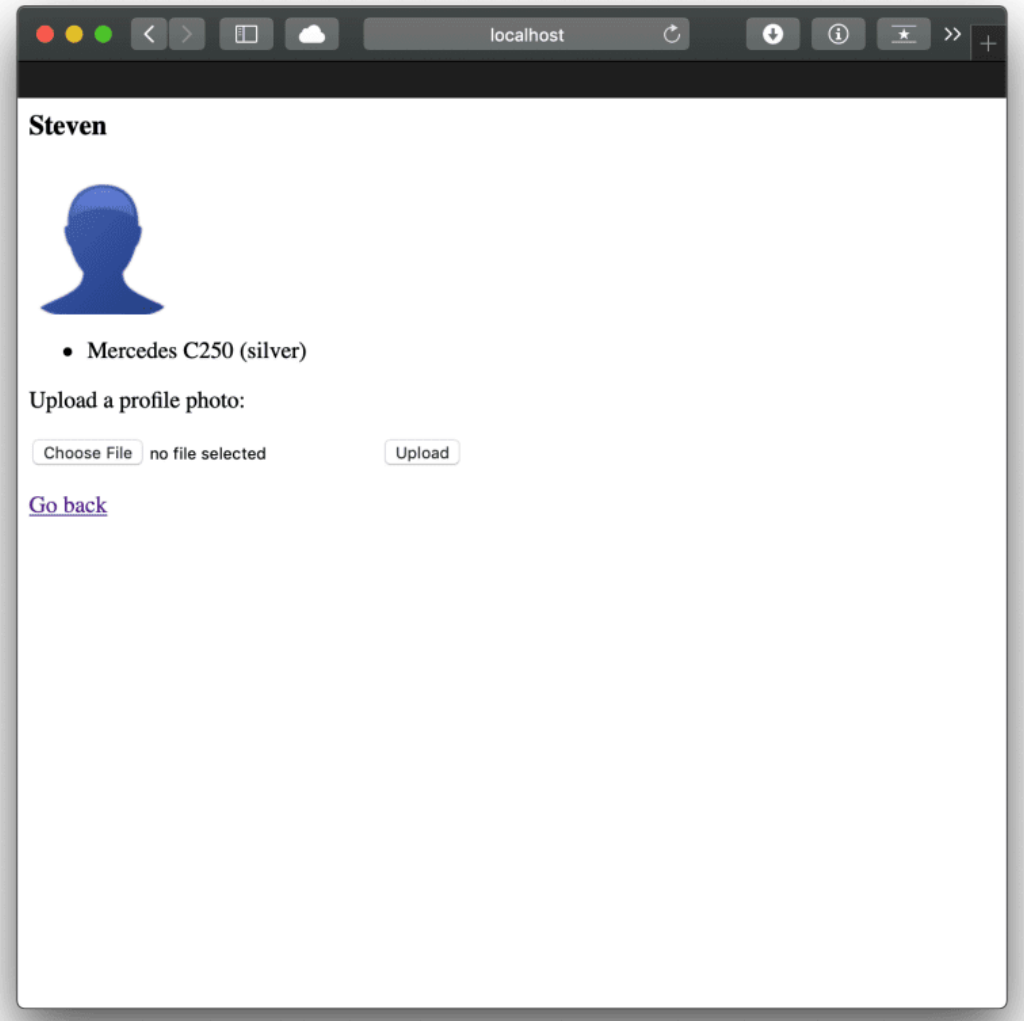
Displaying the image

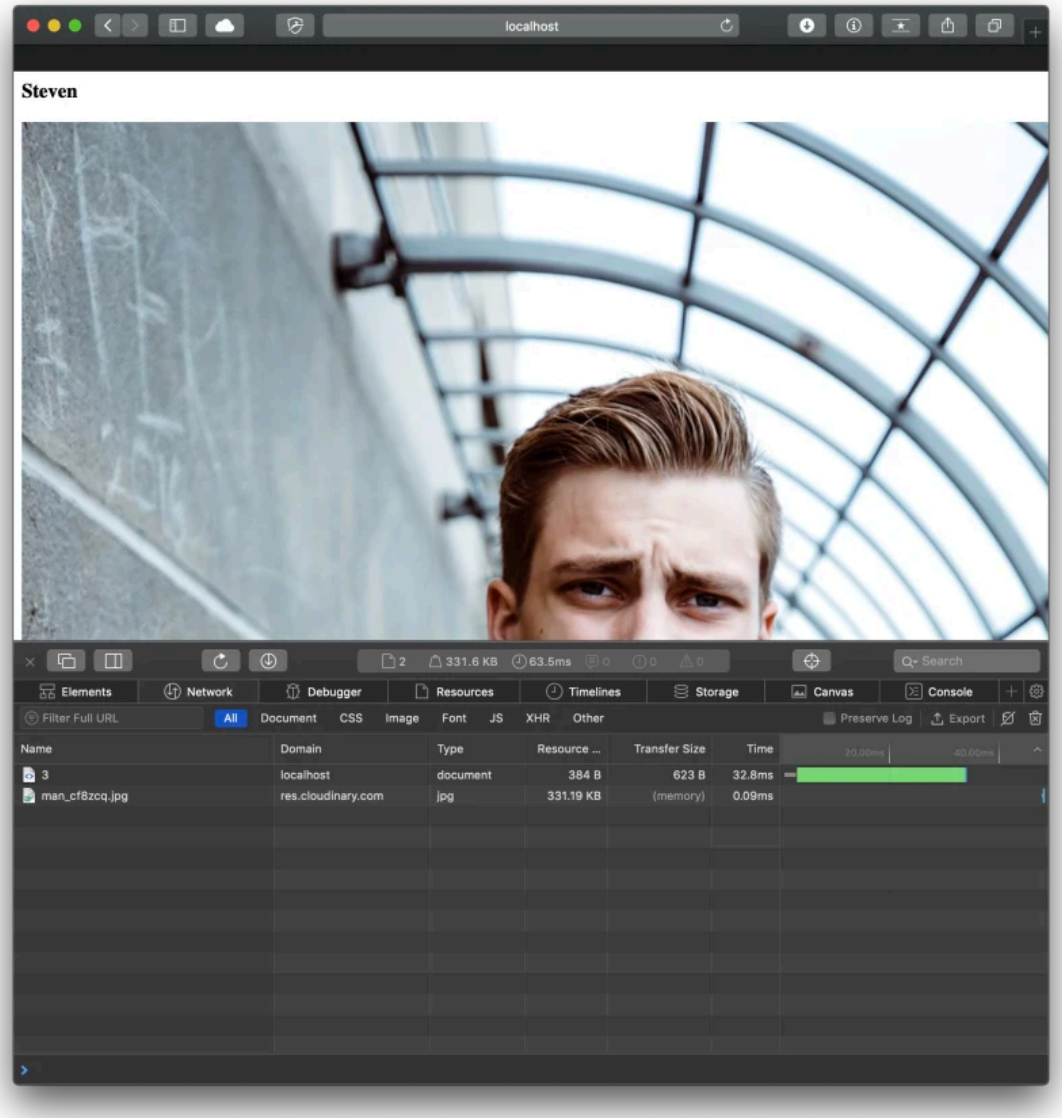
With a photo in place, we can now display it by creating a custom resolver for the photo object:

```
User: {
  photo: (parent, { options }) => {
    let url = cloudinary.url(parent.photo);
    return url;
  }
},
```

The above takes the photo property from the user object and passes that to the **`cloudinary.url()`** method. This method returns a full Cloudinary URL which looks like: **`https://res.cloudinary.com/account-name/image/upload/filename.jpg`**.

The application works at this point. Here are two screenshots of the same user profile - one before and one after the file upload.





Even though the file upload works, there's an obvious problem - the image is quite large! We want to use it as a thumbnail and/or a profile image. Ideally, it should show the users' face and be a lot smaller (both in terms of the actual pixel width as well as the file size).

Applying some CSS to the photo wouldn't decrease the file size - the browser would still have to download the entire image and chew up bandwidth.

Luckily, we have uploaded our image to Cloudinary, and can, therefore, serve it from Cloudinary too. We already saw how to generate a

modified URL via a custom resolver - how about extending that resolver to allow for "options"? Options would be a list of parameters that we can specify via the query mutation that will modify the final Cloudinary URL.

That's a great idea, but how would we pass an "options" object to the resolver? Ideally, we'd like to pass multiple options - like the width of the image, as well as a crop option to create a thumbnail.

The answer to our problem is a custom scalar type in GraphQL.

Custom GraphQL Scalars

What are scalar values in GraphQL? Those would be **`String`**, **`Int`**, **`Boolean`** (and there are a bunch of others as well).

GraphQL allows for the creation of a custom scalar type which is useful in situations where the default scalar types are not robust enough, or when a custom atomic data type is required.

Ideally, the query for the photo should look like:

```
photo(options:"200, false, true, face")
```

It'd be great if we could pass in some options. This means we need to update the type definition (schema) for **`User`**:

```
type User {
  id: ID!
```

```
    name: String!
    photo(options: CloudinaryOptions): String
  }

  scalar CloudinaryOptions
```

Notice we defined `options` as an argument which takes a custom scalar type `CloudinaryOptions` (defined after the `User` definition).

In order to create the custom scalar type we need to import the `GraphQLScalarType` in our resolver:

```
const { GraphQLScalarType } = require('graphql');
```

Then, add the following to the resolver:

```
CloudinaryOptions: new GraphQLScalarType({
  name: 'CloudinaryOptions',
  parseValue(value) {
    return value;
  },
  serialize(value) {
    return value;
  },
  parseLiteral(ast) {
    return ast.value.split(',');
  },
});
```

> *When creating a custom scalar type, we must specify these 3 functions - `parseValue`, `serialize` and `parseLiteral`:*

> 1. *`**parseValue**` is the function that handles the processing of the value from the client.*
> 2. *`**serialize**` is a function responsible for sending the value to the client.*
> 3. *`**parseLiteral**` is invoked to parse the client's input that came from a query.*

We can access the options sent via the query itself via the `**parseLiteral()**` function which returns an array of elements. If we specified the following: `**photo(options:"200, false, true, face")**` in the query - `**parseLiteral()**` will return `**[200, false, true, face]**`.

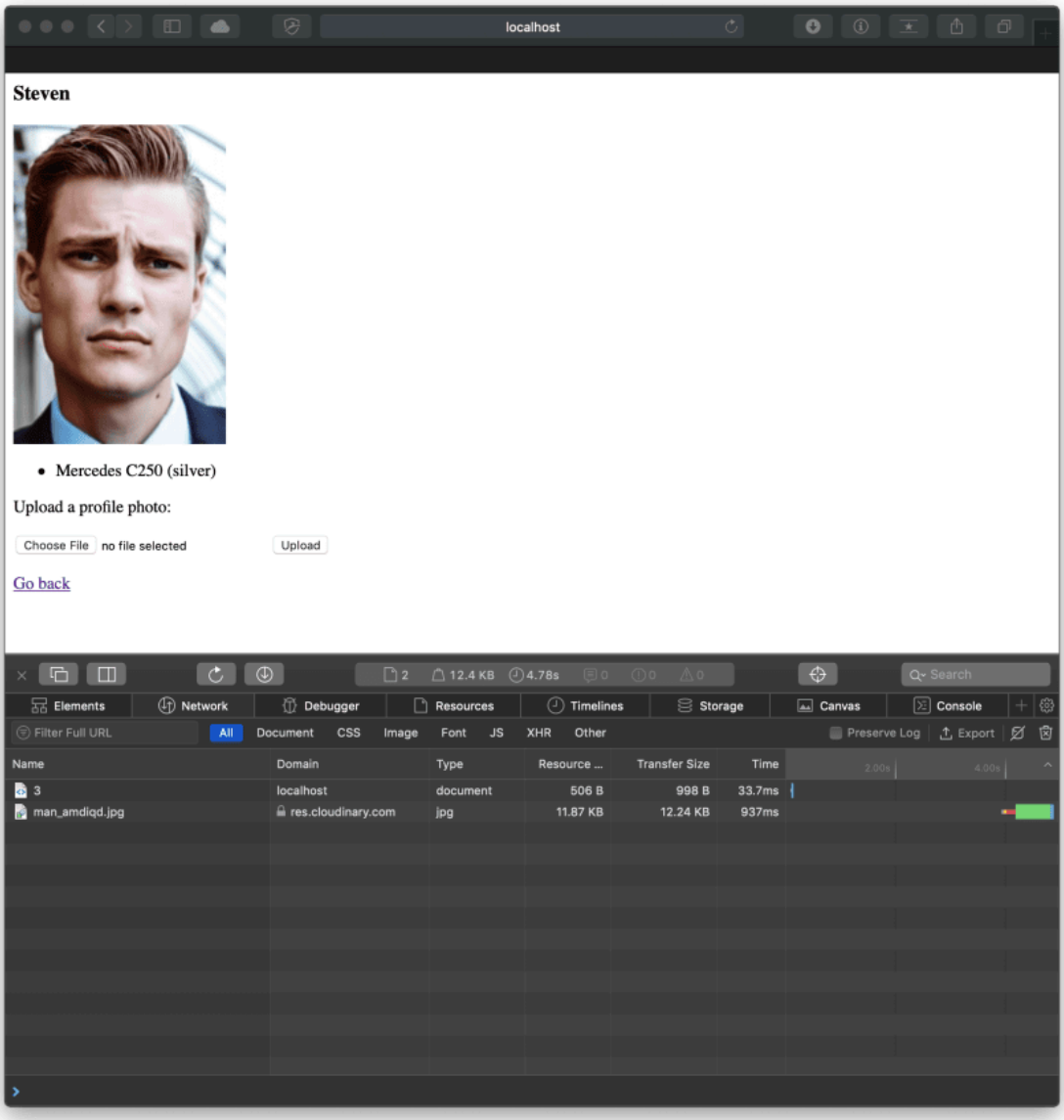
Voilà! We can now access this array in the resolver! This also means we can update the photo resolver to look like:

```
photo: (parent, { options }) => {
  let url = cloudinary.url(parent.photo);
  if (options) {
    // width: Int, q_auto: Boolean, f_auto: Boolean, face
    const [width, q_auto, f_auto, face] = options;
    const cloudinaryOptions = {
      ...(q_auto === 'true' && { quality: 'auto' }),
      ...(f_auto === 'true' && { fetch_format: 'auto' }),
      ...(face && { crop: 'thumb', gravity: 'face' }),
      width,
      secure: true,
    };
    url = cloudinary.url(parent.photo, cloudinaryOptions)
    return url;
  }
```

```
    return url;
  };
```

We take the values from the array and build an options array that we'll send to Cloudinary. Via this options array, calling the **`cloudinary.url()`** returns a modified URL. We also added a **`secure: true`** flag so that the Cloudinary URL will always start with 'https'. This is regardless of which options are sent.

Let's refresh the application and see how the user profile changed. We now get a much smaller image (in both pixel width and file size), with a focus on the face.



Conclusion

In this article we saw how to manage images by a third party service and custom scalar types in GraphQL. There are other alternatives to serving images via GraphQL of course but it makes sense to use a service that already takes care of hosting and the transformation of images giving us a lot more out of the box than just displaying a certain image.
