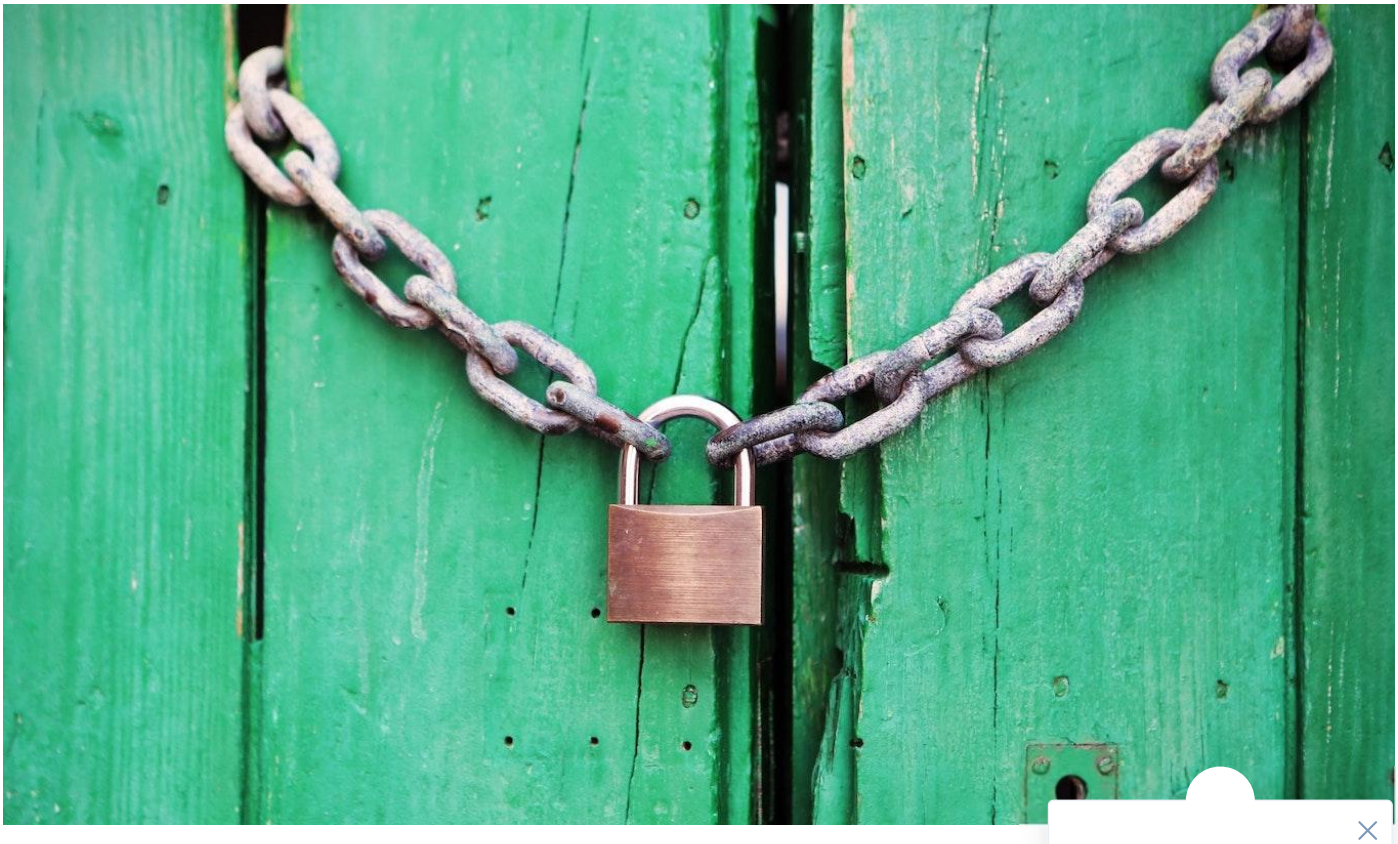
Authorization and Authentication in GraphQL

(https://blog.carbonfive.com/author/linden_melvin/) Posted on 24th March 2020 by Linden Melvin (https://blog.carbonfive.com/author/linden_melvin/) in Development (https://blog.carbonfive.com/category/development/)



Introduction

GraphQL is growing in popularity because it allows applications to request only the data they need using a strongly typed, self-documenting query structure that enables an API to deliver data that can evolve over time.

Unlike traditional REST APIs, GraphQL exposes a single endpoint to query and mutate data. Upon learning this, one of the first questions that comes up for many developers is: “How do I implement authorization and authentication in GraphQL?”

Authorization and authentication in GraphQL can be perplexing if you are a developer coming from a REST API background. GraphQL is a surprisingly thin API layer. The spec (http://spec.graphql.org/) is relatively short and is completely un-opinionated about how authorization and authentication are implemented, leaving the implementation details up to the developer.

Authorization patterns in GraphQL are quite different than in a REST API. GraphQL is not opinionated about how authorization is implemented. To quote directly from graphql.org, (https://graphql.org/learn/authorization/) “Delegate authorization logic to the business logic layer.” It is up to the developer to handle authorization when using GraphQL.

GraphQL is also un-opinionated about how authentication is implemented. Authentication patterns in GraphQL, however, are very similar to patterns used in REST APIs: a user provides login credentials, an authentication token is generated and provided by the client in each subsequent request.

The implementation details for authorization and authentication in GraphQL can be a little tricky at first. With the help of a simple example GraphQL implementation, we can shed some light on how to approach these very important pieces of your API design.

Disclaimer

All examples in this post will be in JavaScript. We will be using Apollo (https://www.apollographql.com/) to get things up and running. All of these concepts are applicable for a custom implementation of GraphQL, but Apollo takes boilerplate configuration out of the equation.

Authorization

Resolvers & Contexts

Resolvers are the building blocks of GraphQL used to connect the schema with the data. These functions define how to fetch data for a query and update data for a mutation. Since resolvers are simple functions that return data, they do not care about the structure of the underlying datastore.

Using Apollo, each resolver is a function that receives four parameters: `root` (or `parent`), `args`, `context`, and `info`. For the purposes of this post, we are interested in the `context` argument. It is the key to handling authorization in our Apollo server. `context` is passed to each resolver and contains data that each resolver can access. The importance of `context` is that it is initialized at the request level, meaning we can use it to store metadata about the user making the request. This is our entry point for implementing authorization.

Walkthrough

Let’s build a basic Apollo server:

```
1
2  const { ApolloServer, gql } = require('apollo-server');
3
4  const resolvers = require("./resolvers");
5
6  const typeDefs = gql`
7    type Post {
8      title: String!
9      authorId: String!
10   }
11
12   type Query {
13     posts: [Post!]!
14   }
15 `;
16
```

```
17  const server = new ApolloServer({
18    typeDefs,
19    resolvers
20  });
21
22  server.listen().then(({ url }) => {
23    console.log(`🚀 Server ready at ${url}`);
24  });
```

base.js view raw (https://gist.github.com/lindenmelvin/89d0c6eecc2bc0daf47ee70b7f537466/raw/1051acf3fa8586aa9682c12a215672aab32745de/base.js) (https://gist.github.com/lindenmelvin/89d0c6eecc2bc0daf47ee70b7f537466#file-base-js) hosted with ♡ by GitHub (https://github.com)

Our inline `typeDefs` defines our schema: there are `Post` objects that have a `title` and an `authoId`, and we have exposed one root query called `posts` which will allow us to fetch a list of `Post` objects.

The resolvers (in `resolvers.js`) are responsible for what it means to "fetch a list of `Post` objects.":

```
1  const posts = require("./posts");
2
3  module.exports = {
4    Query: {
5      posts: () => {
6        return posts;
7      }
8    }
9  };
```

resolvers.js view raw (https://gist.github.com/lindenmelvin/f83e923d32aabbe1a877f081aa9e6751/raw/6b9e58358418d6aa6ca5c8002b13e89e11040ef2/resolvers.js) (https://gist.github.com/lindenmelvin/f83e923d32aabbe1a877f081aa9e6751#file-resolvers-js) hosted with ♡ by GitHub (https://github.com)

The `posts` resolver simply returns the contents of `posts.js`. As mentioned before, GraphQL doesn't care where the data comes from. That means, for the purposes of this simple example, our data can simply be an in-memory array. For example:

```
1  module.exports = [
2    {
3      authorId: 1,
4      title: "Post 1"
5    },
6    {
7      authorId: 1,
8      title: "Post 2"
9    },
10   {
11     authorId: 2,
12     title: "Post 3"
13   },
14   {
15     authorId: 2,
16     title: "Post 4"
17   }
18 ];
```

posts.js view raw (https://gist.github.com/lindenmelvin/52a0d9076f05fa0438174b70ca52fa5d/raw/462364a4f0c1f6c13257b3723f8134dcf7e75679/posts.js) (https://gist.github.com/lindenmelvin/52a0d9076f05fa0438174b70ca52fa5d#file-posts-js) hosted with ♡ by GitHub (https://github.com)

`posts` is an in-memory array of four `Post` objects. As you can see from the `authorId` on the post objects, we have posts from two different authors. Ideally, when a user makes a request for posts, we should only return the posts belonging to that user.

Now that we have the scaffolding for our Apollo server, let's construct a `context` object:

```
1
2  const { ApolloServer, gql } = require('apollo-server');
3
4  const resolvers = require("./resolvers");
```

```
5
6   const typeDefs = gql`
7     type Post {
8       title: String!
9       authorId: String!
10    }
11
12    type Query {
13      posts: [Post!]!
14    }
15  `;
16
17  const context = () => {
18    return {
19      user: { id: 1 }
20    }
21  }
22
23  const server = new ApolloServer({
24    typeDefs,
25    resolvers,
26    context
27  });
28
29  server.listen().then(({ url }) => {
30    console.log(`🚀 Server ready at ${url}`);
31  });
```

no-authentication.js (https://gist.github.com/lindenmelvin/2c32a584ab2ff7919c2bde02380d917c#file-no-authentication-js) view raw (https://gist.github.com/lindenmelvin/2c32a584ab2ff7919c2bde02380d917c/raw/cf0d5f832e772d7cf98213c1f946d775f74b1847/no-authentication.js) hosted with ♡ by GitHub (https://github.com)

We are defining a `context` object and storing a `user` value in it. The user value we are defining here will be provided to each resolver through the `context` object. Let's not worry about authenticating the user just yet. Instead, we will assume that the `user` has an `id` of `1` and that value will be stored in our `context`.

The resolver function for fetching posts is passed four arguments, including the `context` object. Using `context`, we are able to access the current user's `id` and only return posts that belong to that user:

```
1  const posts = require("./posts");
2
3  module.exports = {
4    Query: {
5      posts: (_parent, _args, context, _info) => {
6        return posts.filter(post => post.authorId === context.user.id);
7      }
8    }
9  };
```

resolvers.js (https://gist.github.com/lindenmelvin/93b10c641440551c89a73659c3847ce8#file-resolvers-js) view raw (https://gist.github.com/lindenmelvin/93b10c641440551c89a73659c3847ce8/raw/4442af19816c7cbe789407e5480c9905e1b84597/resolvers.js) hosted with ♡ by GitHub (https://github.com)

This demonstrates how we can leverage the `context` object to have resolver-level visibility into who is trying to access the data.

Of course, we could expand this: instead of simply passing an `id` for the user, we could easily pass a more complex object containing `roles` or `permissions`. For example:

```
1  const posts = require("./posts");
2
3  module.exports = {
4    Query: {
5      posts: (_parent, _args, context, _info) => {
6        return posts.filter(post => {
7          const isAuthor = post.authorId === context.user.id;
8          const isAdmin = context.user.roles.includes("admin");
```

```
          return isAuthor || isAdmin;
        });
      }
    }
  };
```

resolvers-with-roles.js (https://gist.github.com/lindenmelvin/51bf29fa7b329e8268e0e4b5df07eab1#file-resolvers-with-roles-js) hosted with ♡ by GitHub (https://github.com) view raw (https://gist.github.com/lindenmelvin/51bf29fa7b329e8268e0e4b5df07eab1/raw/182b73b495484ff54406b85d1491575c0b6fcede/resolvers-with-roles.js)

If we assume we have a value of `admin` as a possible role, we now have the ability to let a user see a post if they own it OR they are an admin!

Authentication

JWT

With an understanding of how to approach authorization in GraphQL, the next step is to figure out how we can authenticate a user so we can use a real value in the `context` object.

Similar to authorization, GraphQL is not opinionated about how you go about implementing authentication. It is up to the developer to define a system for taking user authentication credentials, verifying them, and giving the user access.

There are plenty of options when it comes to implementing authorization. For this example, we will implement a simple authentication system using JSON Web Tokens (JWT) (https://jwt.io). JWT is used to securely send information between two parties as a JSON object, using a digital signature to verify the information has not been changed and can be trusted.

Walkthrough

Let's start off by updating the resolvers and schemas from before to allow us to authenticate a user.

```
const posts = require("./posts");
const { authenticate } = require("./authService");

module.exports = {
  Query: {
    posts: (_parent, _args, context, _info) => {
      return posts.filter(post => {
        const isAuthor = post.authorId === context.user.id;
        const isAdmin = context.user.roles.includes("admin");
        return isAuthor || isAdmin;
      });
    }
  },
  Mutation: {
    login: (_parent, args, _context, _info) => {
      const { email, password } = args;
      const token = authenticate(email, password);
      return { token };
    }
  }
};
```

resolvers-with-auth.js (https://gist.github.com/lindenmelvin/e3c7463681f79a463f6461c489ebeeb9#file-resolvers-with-auth-js) hosted with ♡ by GitHub (https://github.com) view raw (https://gist.github.com/lindenmelvin/e3c7463681f79a463f6461c489ebeeb9/raw/5e90e7271d68ab24edc7075604f89ac56c98d905/resolvers-with-auth.js)

We have added a new Mutation called `login` that will allow us to handle login credentials provided by the client. To help handle the authentication logic, we have created an `auth-service`:

```
const jwt = require('jsonwebtoken');

module.exports = {
  authenticate: (email, password) => {
    let user;

```

```
    // Implement the credential validation however you'd like.
    if (email.length && password.length) {
      user = {
        id: 1,
        roles: ["admin"]
      }
    }

    if (!user) throw new Error("Invalid credentials.");

    return jwt.sign(user, process.env.JWT_SECRET);
  },
  validateToken: token => {
    try {
      const { id, roles } = jwt.verify(token, process.env.JWT_SECRET);
      return { id, roles };
    } catch (e) {
      throw new Error('Authentication token is invalid.');
    }
  }
}
```

auth-service.js view raw (https://gist.github.com/lindenmelvin/0c11a5d4a13029561c20e5b2bf54bcb6/raw/a51f3da8cf271519aea86237dd1387d1e986e618/auth-service.js) (https://gist.github.com/lindenmelvin/0c11a5d4a13029561c20e5b2bf54bcb6#file-auth-service-js) hosted with ♡ by GitHub (https://github.com)

The `auth-service` uses JWT to generate a token that contains the `id` and `roles` of the authenticated user and that can be handed down to the client to stored in the `Authorization` header and be used in subsequent requests. For this example, the actual authentication logic is trivial, simply checking that the email and password values are not empty. This logic can be updated to fit your authentication needs.

Next, let's look at the updated `server` file:

```
const { ApolloServer, gql } = require('apollo-server');

const { validateToken } = require("./authService");
const resolvers = require("./resolvers");

const typeDefs = gql`
  type User {
    id: ID!
    email: String!
    password: String!
  }
  type Post {
    title: String!
    authorId: ID!
  }
  type AuthResponse {
    token: String!
  }
  type Query {
    posts: [Post!]!
  }
  type Mutation {
    login(email: String!, password: String!): AuthResponse!
  }
`;

const context = ({ req }) => {
  if (req.body.query.match("Login")) return {};

  const authorizationHeader = req.headers.authorization || '';
  const token = authorizationHeader.split(' ')[1];
```

```
32
33    if (!token) throw new Error("Authentication token is required.");
34
35    const user = validateToken(token);
36
37    return { user };
38  }
39
40  const server = new ApolloServer({
41    typeDefs,
42    resolvers,
43    context,
44    playground: false
45  });
46
47  server.listen().then(({ url }) => {
48    console.log(`🚀 Server ready at ${url}`);
49  });
```

server.js view raw (https://gist.github.com/lindenmelvin/627f899fcd785e4f0027690b6d3d1529/raw/888f1c1231e70b1bf57ef93675f58ecbd32a020e/server.js) (https://gist.github.com/lindenmelvin/627f899fcd785e4f0027690b6d3d1529#file-server-js) hosted with ♡ by GitHub (https://github.com)

As we saw in the resolvers, there is a new mutation available to us: `login` which will return the authenticated JWT.

The function responsible for constructing the `context` object has access to the request via the `req` parameter. This allows the context construction function to access the `authorization` header, grab the token, handle the case when the token does not exist, and then validate the token allowing us to get the user data encoded in it. Now that we have access to an authenticated user's data, we can use this during authorization to make sure we are only returning data the user is allowed to see.

Using the API

We have created a query in our Apollo server that a client can use to fetch posts. First, though, the user must log in. Using the JavaScript HTTP client library axios (https://github.com/axios/axios), let's take a look at how to dispatch a request to authenticate using our GraphQL API.

```
1   const axios = require("axios");
2
3   axios.defaults.baseURL = "http://localhost:4000/";
4
5   const login = async (email, password) => {
6     const axiosResponse = await axios.post('/', {
7       query: `mutation Login {
8         login(email: "${email}", password: "${password}") {
9           token
10        }
11      }`
12    });
13
14    const graphqlQueryData = axiosResponse.data;
15    const authenticationToken = graphqlQueryData.data.login.token;
16
17    axios.defaults.headers.common['Authorization'] = `Bearer ${authenticationToken}`;
18  }
19
20  const posts = async () => {
21    const axiosResponse = await axios.post('/', {
22      query: `query FetchPosts {
23        posts {
24          title
25        }
26      }`
27    });
28
29    const graphqlQueryData = axiosResponse.data;
```

```
30    const posts = graphqlQueryData.data.posts;
31
32    return posts;
33  }
34
35  const main = async () => {
36    await login("foo@example.com", "bar");
37    await posts();
38  }
39
40  main();
```

client-auth.js view raw (https://gist.github.com/lindenmelvin/6f201b569d17859b0e37b40e3c7cc5c0/raw/bd1a00df8e85d01f0a4278a034c2e9438341e122/client-auth.js) (https://gist.github.com/lindenmelvin/6f201b569d17859b0e37b40e3c7cc5c0#file-client-auth-js) hosted with ♡ by GitHub (https://github.com)

The `login` method is used to dispatch a query to the Apollo API, providing the email and password from the user. The response contains a JWT token that we add to the axios client default headers for each subsequent request. When we then make a request for posts in the `posts` method, the response will contain all of the posts the user is authorized to see.

Conclusion

Authorization and authentication are fundamentally important pieces of API design. With many developers coming from a REST API background, making the leap to GraphQL can be confusing at first. This confusion stems from the fact that implementing authorization and authentication in GraphQL is left up to the developer. Using the example code provided in this post, you can create a fully functional GraphQL server using Apollo, complete with authorization and authentication.

The complete and functional example files are available here:

Setup

- Clone all files into a directory
- Run `yarn`
- Run `export JWT_SECRET="your_jwt_secret_key"`
- Run `node server.js`
- Run `node client-auth.js`
- See a console log lists of posts fetched from GraphQL

README.md view raw (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839/raw/5923300bfdc914426323de79a46319caaac2f75f/README.md) (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839#file-readme-md) hosted with ♡ by GitHub (https://github.com)

```
1   const jwt = require('jsonwebtoken');
2
3   module.exports = {
4     authenticate: (email, password) => {
5       let user;
6
7       // Implement the credential validation however you'd like.
8       if (email.length && password.length) {
9         user = {
10          id: 1,
11          roles: ["admin"]
12        }
13      }
14
15      if (!user) throw new Error("Invalid credentials.");
16
17      return jwt.sign(user, process.env.JWT_SECRET);
```

```
  },
  validateToken: token => {
    try {
      const { id, roles } = jwt.verify(token, process.env.JWT_SECRET);
      return { id, roles };
    } catch (e) {
      throw new Error('Authentication token is invalid.');
    }
  }
}
```

auth-service.js view raw (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839/raw/5923300bfdc914426323de79a46319caaac2f75f/auth-service.js) (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839#file-auth-service-js) hosted with ♡ by GitHub (https://github.com)

```
const axios = require("axios");

axios.defaults.baseURL = "http://localhost:4000/";

const login = async (email, password) => {
  const axiosResponse = await axios.post('/', {
    query: `mutation Login {
      login(email: "${email}", password: "${password}") {
        token
      }
    }`
  });

  const graphqlQueryData = axiosResponse.data;
  const authenticationToken = graphqlQueryData.data.login.token;

  axios.defaults.headers.common['Authorization'] = `Bearer ${authenticationToken}`;
}

const posts = async () => {
  const axiosResponse = await axios.post('/', {
    query: `query FetchPosts {
      posts {
        title
      }
    }`
  });

  const graphqlQueryData = axiosResponse.data;
  const posts = graphqlQueryData.data.posts;

  return posts;
}

const main = async () => {
  await login("foo@example.com", "bar");
  const postsForUser = await posts();
  console.log("posts", postsForUser)
}

main();
```

client-auth.js view raw (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839/raw/5923300bfdc914426323de79a46319caaac2f75f/client-auth.js) (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839#file-client-auth-js) hosted with ♡ by GitHub (https://github.com)

```
{
  "name": "graphql-authorization-and-authentication",
  "version": "1.0.0",
  "main": "server.js",
  "license": "MIT",
  "dependencies": {
    "apollo-server": "^2.9.16",
    "axios": "^0.19.2",
```

```
    "jsonwebtoken": "^8.5.1"
  }
}
```

package.json view raw (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839/raw/5923300bfdc914426323de79a46319caaac2f75f/package.json) (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839#file-package-json) hosted with ♡ by GitHub (https://github.com)

```
module.exports = [
  {
    authorId: 1,
    title: "Post 1"
  },
  {
    authorId: 1,
    title: "Post 2"
  },
  {
    authorId: 2,
    title: "Post 3"
  },
  {
    authorId: 2,
    title: "Post 4"
  }
];
```

posts.js view raw (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839/raw/5923300bfdc914426323de79a46319caaac2f75f/posts.js) (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839#file-posts-js) hosted with ♡ by GitHub (https://github.com)

```
const posts = require("./posts");
const { authenticate } = require("./auth-service");

module.exports = {
  Query: {
    posts: (_parent, _args, context, _info) => {
      return posts.filter(post => {
        const isAuthor = post.authorId === context.user.id;
        const isAdmin = context.user.roles.includes("admin");
        return isAuthor || isAdmin;
      });
    }
  },
  Mutation: {
    login: (_parent, args, _context, _info) => {
      const { email, password } = args;
      const token = authenticate(email, password);
      return { token };
    }
  }
};
```

resolvers.js view raw (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839/raw/5923300bfdc914426323de79a46319caaac2f75f/resolvers.js) (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839#file-resolvers-js) hosted with ♡ by GitHub (https://github.com)

```
const { ApolloServer, gql } = require('apollo-server');

const { validateToken } = require("./auth-service");
const resolvers = require("./resolvers");

const typeDefs = gql`
  type User {
    id: ID!
    email: String!
    password: String!
  }

  type Post {
    title: String!
```

```
    authorId: ID!
  }

  type AuthResponse {
    token: String!
  }

  type Query {
    posts: [Post!]!
  }

  type Mutation {
    login(email: String!, password: String!): AuthResponse!
  }
`;

const context = ({ req }) => {
  if (req.body.query.match("Login")) return {};

  const authorizationHeader = req.headers.authorization || '';
  const token = authorizationHeader.split(' ')[1];

  if (!token) throw new Error("Authentication token is required.");

  const user = validateToken(token);

  return { user };
}

const server = new ApolloServer({
  typeDefs,
  resolvers,
  context,
  playground: false
});

server.listen().then(({ url }) => {
  console.log(`🚀 Server ready at ${url}`);
});
```

server.js view raw (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839/raw/5923300bfdc914426323de79a46319caaac2f75f/server.js) (https://gist.github.com/lindenmelvin/a0ac0d571eecc65838fc90a191c81839#file-server-js) hosted with ♡ by GitHub (https://github.com)

Interested in more software development tips & insights? Visit the development section (https://blog.carbonfive.com/category/development/) on our blog!

Related Posts

- An AJAX Auto-Save Implementation (https://blog.carbonfive.com/an-ajax-auto-save-implementation/)
- The Best of Both Worlds: HTML Apps & Svelte (https://blog.carbonfive.com/the-best-of-both-worlds-html-apps-svelte/)
- Shifting to A Work From Home Policy: A Mini Survival Guide for Your Company (https://blog.carbonfive.com/shifting-to-a-work-from-home-policy-a-mini-survival-guide-for-your-company/)

(https://twitter.com/intent/tweet?status=Authorization+and+Authentication+in+GraphQL+https%3A%2F%2Fblog.carbonfive.com%2Fauthorization-and-authentication-in-graphql%2F)

Carbon Five is a full service software consultancy that helps startups and established organizations design, build, and ship awesome products. If you have a project you'd like us to take a look at, or are interested in joining our team, please let us know (http://www.carbonfive.com/contact).