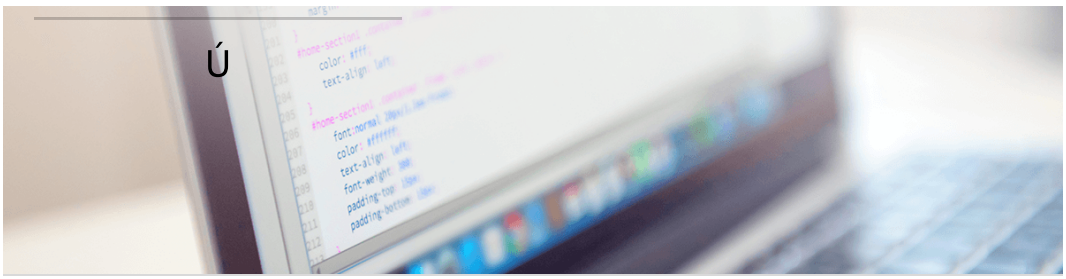
An Introduction to GraphQL: Authentication

by Peter Mbanugo | Á August 24, 2020 | Developer Central, Web | À 0 Comments



Ú

É	ċ
á	ö

The GraphQL specification that defines a type system, query and schema language for your Web API, and an execution algorithm for how a GraphQL service (or engine), should validate and execute queries against the GraphQL schema. In this article, you'll learn how to implement authentication in a GraphQL server.

GraphQL, described as a data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data, allows varying clients to use your API and query for just the data they need. It helps solve some performance issues that some REST services have—over-fetching and under-fetching. The GraphQL specification defines a type system, query language, and schema language for your Web API, and an execution algorithm for how a GraphQL service (or engine) should validate and execute queries against the GraphQL schema.

There are different ways to handle authentication in a GraphQL server, and in this post, I' ll walk you through building a signup and signin resolvers, then building a wrapper function that will be used to wrap specific resolvers for the root fields we want to only be accessible to authenticated users.

We will be working with an existing GraphQL server—adding new resolvers to it and protecting existing resolvers. If you followed along from previous articles before this one, you should be familiar with the project and probably already have the code from where we stopped in the last article, An Introduction to GraphQL: Subscriptions.

If you don' t already have this project, but want to code along, download the project from GitHub, and copy the files from src-part-3 folder to the main src folder. Then follow the instructions in the **README** file to set up the project.

Allow signup and signin

We will be adding two new operations to the schema: one for users to sign up, and another for sign-in. We will store the user information in the database; therefore, we need to

update the database model. Open the file src/prisma/datamodel.prisma and add the model below to it.


```
type User {
  id: ID! @id
  name: String!
  email: String! @unique
  password: String!
}
```

The User model represents the user who needs to be authenticated to use the API, and we will store this information in the database. After updating the datamodel, we need to update the Prisma server with this change. Open the terminal and switch to the src/prisma directory and run primsa deploy.

```
Deploying service `graphql-intro` to stage `dev` to server `prisma-us1` 3.3s

Changes:

  User (Type)
  + Created type `User`
  + Created field `id` of type `ID!`
  + Created field `name` of type `String!`
  + Created field `email` of type `String!`
  + Created field `password` of type `String!`

Applying changes 1.5s

Your Prisma GraphQL database endpoint is live:
```

When this completes successfully, run the command prisma generate to update the auto-generated prisma client.

Update the GraphQL Schema

With our datamodel updated, we will now update the GraphQL schema with two new root fields on the Mutation type. Open src/index.js and add two new root fields, signup and signin, to the Mutation type.

```
signup(email: String!, password: String!, name: String!): Aut
signin(email: String!, password: String!): AuthPayload
```

These mutations will be used for signup and signin requests, and will return data of type AuthPayload. Go ahead and add definition for this new type to the schema:

```
type AuthPayload {
  token: String!
  user: User!
}

type User {
  id: ID!
```

```
  name: String!
  email: String!
}
```


With those new changes, your schema definition should match what you see below:

```
const typeDefs = `
type Book {
    id: ID!
    title: String!
    pages: Int
    chapters: Int
    authors: [Author!]!
}

type Author {
    id: ID!
    name: String!
    books: [Book!]!
}

type Query {
  books: [Book!]
  book(id: ID!): Book
  authors: [Author!]
}

type Mutation {
  book(title: String!, authors: [String!]!, pages: Int, chapters:
  signup(email: String!, password: String!, name: String!): Au
  signin(email: String!, password: String!): AuthPayload
}

type Subscription {
  newBook(containsTitle: String): Book!
}

type AuthPayload {
  token: String!
  user: User!
}

type User {
  id: ID!
  name: String!
  email: String!
}
`;
```

Implementing the Resolvers

Now that we have added new types and extended the Mutation type, we need to implement resolver functions for them. Open src/index.js, go to line 82 where we can add resolver functions for the Mutation root fields and paste in the code below:

```
signup: async (root, args, context, info) => {
  const password = await bcrypt.hash(args.password, 10);
  const user = await context.prisma.createUser({ ...args, pas
  const token = jwt.sign({ userId: user.id }, APP_SECRET);

  return {
    token,
    user
  };
},
signin: async (root, args, context, info) => {
  const user = await context.prisma.user({ email: args.email
  if (!user) {
    throw new Error("No such user found");
  }
  const valid = await bcrypt.compare(args.password, user.p
  if (!valid) {
    throw new Error("Invalid password");
  }

  const token = jwt.sign({ userId: user.id }, APP_SECRET);

  return {
    token,
    user
  };
}
```

The code you just added will handle signup and signin for the application. We used two libraries bcryptjs and jsonwebtoken (which we'll add later) to encrypt the password, and handle token creation and validation. In the signup resolver, the password is hashed before saving the user data to the database. Then we use the jsonwebtoken library to generate a JSON web token by calling jwt.sign() with the app secret used to sign the token. We will add in the APP_SECRET later. The signin resolver validates the email and password. If it's correct, it signs a token and return an object that matches the AuthPayLoad type, which is the return type for the signup and signin mutations.

I'd like to point out that I intentionally skipped adding an expiration time to the generated token. This means that the token a client gets will be used at any time to access the API. In a production app, I'd advise you add an expiration period for the token and validate that in the server.

While we have index.js open, add the code statement below after line 2:

```
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const APP_SECRET = "GraphQL-Vue-React";
```

Now open the command line and run the command below to install the needed dependencies.

```
npm install --save jsonwebtoken bcryptjs
```


Requiring Authentication for the API

So far we have implemented a mechanism for users to signin and get token that' ll be used to validate them as a user. We' re now going to move to a new requirement for the API. Which is:

> Only authenticated users should call the book mutation operation.

We will implement this by validating the token from the request. We' ll be using a login token in an HTTP authorization header. Once validated, we check that the user ID from the token matches a valid user in the database. If valid, we put the user object in the context argument that the resolver functions will receive.

Let' s start by putting the user object in the context. Open src/index.js and go to line 129 where the GraphQL server is being initialized. Update the context field to the following:

```
context: async ({ request }) => {
  let user;
  let isAuthenticated = false;
  // get the user token from the headers
  const authorization = request.get("Authorization");
  if (authorization) {
    const token = authorization.replace("Bearer ", "");
    // try to retrieve a user with the token
    user = await getUser(token);
    if (user) isAuthenticated = true;
  }

  // add the user and prisma client to the context
  return { isAuthenticated, user, prisma };
};
```

Before now, we' ve mapped an object that included the prisma client to context. This time around we' re giving it a function and this function will be used to build the context object which every resolver function receives. In this function, we' re getting the token from the request header and passing it to the function getUser(). Once resolved, we return an object that includes the prisma client, the user object, and an additional field used to check if the request is authenticated.

We' re going to define the getUser function which was used earlier in index.js to have the signature below:

```
async function getUser(token) {
  const { userId } = jwt.verify(token, APP_SECRET);
  return await prisma.user({ id: userId });
}
```


Our next step will be to define a wrapper function which will be used to wrap the resolvers we want to be authenticated. This function will use info from the context object to determine access to a resolver. Add this new function in src/index.js.

```
function authenticate(resolver) {
  return function(root, args, context, info) {
    if (context.isAuthenticated) {
      return resolver(root, args, context, info);
    }
    throw new Error(`Access Denied!`);
  };
}
```

What this function checks is if the user is authenticated. If they're authenticated, it'll call the resolver function passed to it. If they're not, it'll throw an exception.

Now go to the book resolver function and wrap it with the authenticate function.

```
book: authenticate(async (root, args, context, info) => {
    let authorsToCreate = [];
    let authorsToConnect = [];

    for (const authorName of args.authors) {
      const author = await context.prisma.author({ name: au
      if (author) authorsToConnect.push(author);
      else authorsToCreate.push({ name: authorName });
    }

    return context.prisma.createBook({
      title: args.title,
      pages: args.pages,
      chapters: args.chapters,
      authors: {
        create: authorsToCreate,
        connect: authorsToConnect
      }
    });
  }),
```

Testing the Application

Now, we're set to test the authentication flow we added to the API. Go ahead and open the command line to the root directory of your project. Run node src/index.js to start the server and go to http://localhost:4000 in the browser.

Run the following query to create a new book:

```
mutation{
  book(title: "GRAND Stack", authors: ["James Blunt"]){
    title
  }
}
```

You should get the error message Access Denied! as a response. We need a token to be able to run that operation. We'll signup a new user and use the returned token in the authorization header.

Run the following query to create a new user:

```
mutation{
  signup(email: "test@test.com", name: "Test account", pas
    token
  }
}
```

It'll run the mutation and return a token. Open the **HTTP HEADERS** pane at the bottom-left corner of the playground and specify the Authorization header as follows:

```
{
  "Authorization": "Bearer __TOKEN__"
}
```

Replace __TOKEN__ with the token in the response you got from the last mutation query. Now re-run the query to create a new book.

```
mutation {
  book(title: "GRAND Stack", authors: ["James Blunt"]){
    title
  }
}
```

This time around we get a response with the title of the book.

That' s a Wrap!

Woohoo! We now have a real-time API that allows for CRUD operations and requires clients to be authenticated to perform some operations. We built our own authentication system by storing user information in the database and encrypting the password using bscriptjs. The context object, which is passed down to every resolver, now includes new properties to determine if the request is authenticated and

also a user object. You can access the user object from any resolver and you may need it to store more information (e.g adding a new property to determine which user created or updated the book data). We added a wrapper function that you can use to wrap any resolver that allows access only to authenticated users. This approach to using a wrapper function is similar to using a middleware. I'll go into more details on GraphQL middleware in a future post.

I hope you've enjoyed reading this. Feel free to drop any questions in the comments. You can find the code on GitHub.

Happy Coding! 🚀

ą Tutorial, web development, graphql, authentication, API

ABOUT THE AUTHOR

Peter Mbanugo

Peter Mbanugo is a software developer, tech writer, and maker of Hamoni Sync. When he's not building tech products, he spends his time learning and sharing his knowledge on topics related to GraphQL, Offline-First and recently WebAssembly. He's also a contributor to Hoodie and a member of the Offline-First community. You can follow him on Twitter.

RELATED POSTS

DEVELOPER CENTRAL WEB

GraphQL: Mutation and Database Access

WEB DEVELOPER CENTRAL

GraphQL: Schema Resolvers, Type System, Schema Language, and Query Language

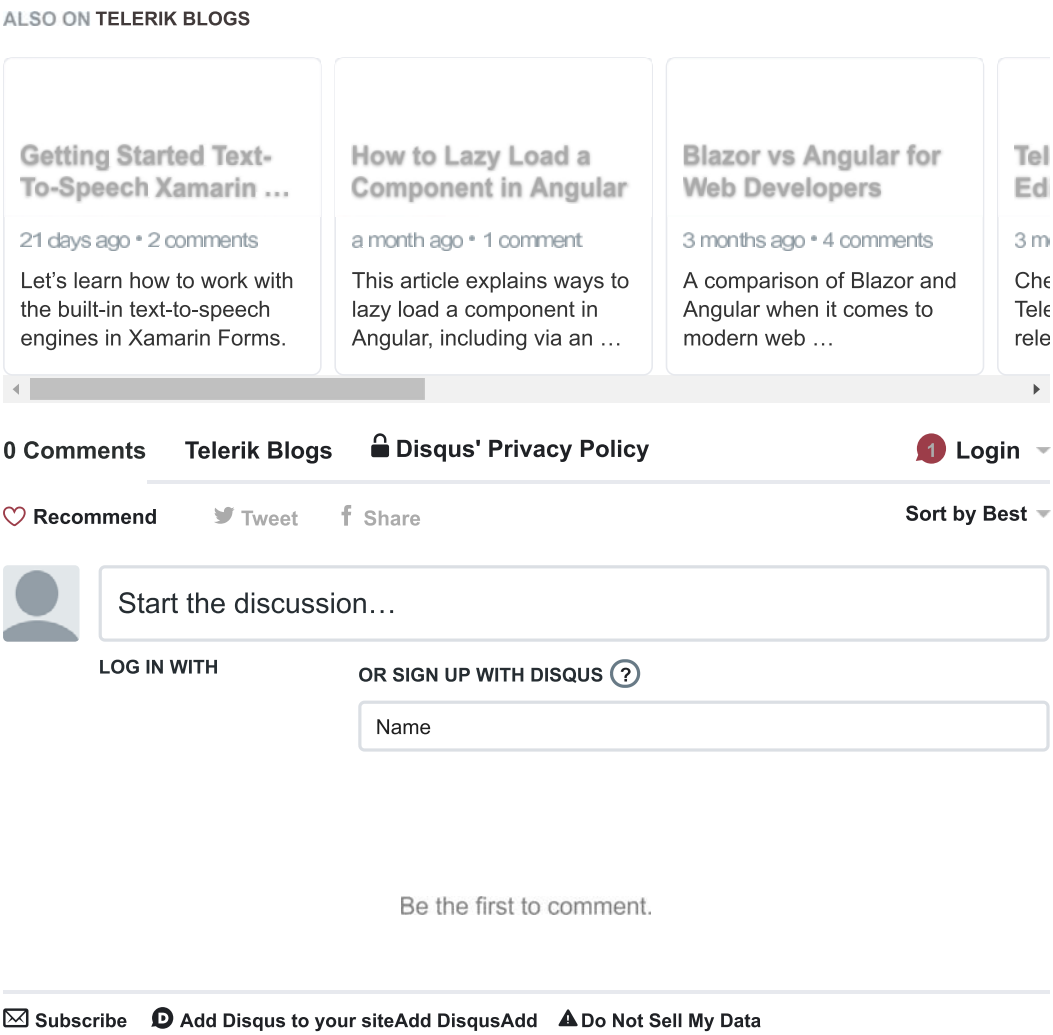
WEB DEVELOPER CENTRAL

An Introduction to GraphQL: Subscriptions

Latest Stories in Your Inbox

Ú

COMMENTS



TECHNOLOGIES

.NET

JavaScript

TOPICS

Developer Central

Web