Authentication

In this section, you're going to implement signup and login functionality that allows your users to authenticate against your GraphQL server.

Adding a `User` model

The first thing you need is a way to represent user data in the database. To do so, you can add a `User` type to your Prisma data model.

You'll also want to add a *relation* between the `User` and the existing `Link` type to express that `Link` s are *posted* by `User` s.

Open `prisma/schema.prisma` and add the following code, making sure to also update your existing `Link` model accordingly:

.../hackernews-node/prisma/data model.prisma

```
model Link {
  id          Int      @id @default(autoincrement())
  createdAt   DateTime @default(now())
  description String
  url         String
  postedBy    User?    @relation(fields: [postedById], references: [id])
  postedById  Int?
}

model User {
  id        Int      @id @default(autoincrement())
  name      String
  email     String   @unique
  password  String
  links     Link[]
}
```

Understanding relation fields

Notice how you're adding a new *relation field* called `postedBy` to the `Link` model that points to a `User` instance. The `User` model then has a `links` field that's a list of `Link` s.

To do this, we need to also define the relation by annotating the `postedBy` field with the `@relation` attribute. This is required for every relation field in your Prisma schema, and all you're doing is defining what the foreign key of the related table will be. So in this case, we're adding an extra field to store the `id` of the `User` who posts a `Link` , and then telling Prisma that `postedById` will be equal to the `id` field in the `User` table.

If this is quite new to you, don't worry! We're going be adding a few of these relational fields and you'll get the hang of it as you go! For a deeper dive on relations with Prisma, check out these docs.

Updating Prisma Client

This is a great time to refresh your memory on the workflow we described for your project at the end of chapter 4!

After every change you make to the data model, you need to migrate your database and then re-generate Prisma Client.

In the root directory of the project, run the following command:

$ .../hackernews-node

```
npx prisma migrate save --experimental
```

In the root directory of the project, run the following command:

$ .../hackernews-node

```
npx prisma migrate save --name "add-user-model" --experimental
```

Now it's time to apply that migration to your database:

$ .../hackernews-node

```
npx prisma migrate up --experimental
```

Your database structure should now be updated to reflect the changes to your data model.

Finally, you need to re-generate PrismaClient.

Run the following command:

$ .../hackernews-node

```
npx prisma generate
```

That might feel like a lot of steps, but the workflow will become automatic by the end of this tutorial!

Your database is ready and Prisma Client is now updated to expose all the CRUD queries for the newly added `User` model – woohoo! 🎉

Extending the GraphQL schema

Remember back when we were setting up your GraphQL server and discussed the process of schema-driven development? It all starts with extending your schema definition with the new operations that you want to add to the API - in this case a `signup` and `login` mutation.

Open the application schema in `src/schema.graphql` and update the `Mutation` type

```
type Mutation {
  post(url: String!, description: String!): Link!
  signup(email: String!, password: String!, name: String!): AuthPayload
  login(email: String!, password: String!): AuthPayload
}
```

Next, go ahead and add the `AuthPayload` along with a `User` type definition to the file.

Still in `src/schema.graphql`, add the following type definitions:

…/hackernews-node/src/schema.graphql

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

type User {
  id: ID!
  name: String!
  email: String!
  links: [Link!]!
}
```

The `signup` and `login` mutations behave very similarly: both return information about the `User` who's signing up (or logging in) as well as a `token` which can be used to authenticate subsequent requests against your GraphQL API. This information is bundled in the `AuthPayload` type.

Finally, you need to reflect that the relation between `User` and `Link` should be bi-directional by adding the `postedBy` field to the existing `Link` model definition in `schema.graphql`:

…/hackernews-node/src/schema.graphql

```
  url: String!
  postedBy: User
}
```

Implementing the resolver functions

After extending the schema definition with the new operations, you need to implement resolver functions for them. Before doing so, let's actually refactor your code a bit to keep it more modular!

You'll pull out the resolvers for each type into their own files.

First, create a new directory called `resolvers` and add four files to it: `Query.js`, `Mutation.js`, `User.js` and `Link.js`. You can do so with the following commands:

$ .../hackernews-node

```
mkdir src/resolvers
touch src/resolvers/Query.js
touch src/resolvers/Mutation.js
touch src/resolvers/User.js
touch src/resolvers/Link.js
```

Next, move the implementation of the `feed` resolver into `Query.js`.

In `Query.js`, add the following function definition:

.../hackernews-node/src/resolvers/Query.js

```
function feed(parent, args, context, info) {
  return context.prisma.link.findMany()
}

module.exports = {
  feed,
}
```

This is pretty straighforward. You're just reimplementing the same functionality from before with a dedicated function in a different file. The `Mutation` resolvers are next.

Adding authentication resolvers

Open `Mutation.js` and add the new `login` and `signup` resolvers (you'll add the `post` resolver in a bit):

.../hackernews-node/src/resolvers/Mutation.js

```
async function signup(parent, args, context, info) {
  // 1
  const password = await bcrypt.hash(args.password, 10)

  // 2
  const user = await context.prisma.user.create({ data: { ...args, password }

  // 3
  const token = jwt.sign({ userId: user.id }, APP_SECRET)

  // 4
  return {
    token,
    user,
  }
}

async function login(parent, args, context, info) {
  // 1
  const user = await context.prisma.user.findOne({ where: { email: args.email
  if (!user) {
    throw new Error('No such user found')
  }

  // 2
  const valid = await bcrypt.compare(args.password, user.password)
  if (!valid) {
    throw new Error('Invalid password')
  }

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

  // 3
  return {
    token,
    user,
  signup,
  login,
  post,
}
```

Let's use the good ol' numbered comments again to understand what's going on here – starting with `signup`.

1. In the `signup` mutation, the first thing to do is encrypt the `User`'s password using the `bcryptjs` library which you'll install soon.
2. The next step is to use your `PrismaClient` instance (via `prisma` as we covered in the steps about `context`) to store the new `User` record in the database.
3. You're then generating a JSON Web Token which is signed with an `APP_SECRET`. You still need to create this `APP_SECRET` and also install the `jwt` library that's used here.
4. Finally, you return the `token` and the `user` in an object that adheres to the shape of an `AuthPayload` object from your GraphQL schema.

Now on the `login` mutation!

1. Instead of *creating* a new `User` object, you're now using your `PrismaClient` instance to retrieve an existing `User` record by the `email` address that was sent along as an argument in the `login` mutation. If no `User` with that email address was found, you're returning a corresponding error.
2. The next step is to compare the provided password with the one that is stored in the database. If the two don't match, you're returning an error as well.
3. In the end, you're returning `token` and `user` again.

Let's go and finish up the implementation.

First, add the required dependencies to the project:

$ .../hackernews-node/

```
npm install jsonwebtoken bcryptjs
```

Next, you'll create a few utilities that are being reused in a few places.

Create a new file inside the `src` directory and call it `utils.js`:

$ .../hackernews-node/

```
touch src/utils.js
```

Now, add the following code to it:

.../hackernews-node/src/utils.js

```
const jwt = require('jsonwebtoken')
const APP_SECRET = 'GraphQL-is-aw3some'

function getUserId(context) {
  const Authorization = context.request.get('Authorization')
  if (Authorization) {
    const token = Authorization.replace('Bearer ', '')
    const { userId } = jwt.verify(token, APP_SECRET)
    return userId
  }

  throw new Error('Not authenticated')
}

module.exports = {
  APP_SECRET,
  getUserId,
}
```

The `APP_SECRET` is used to sign the JWTs which you're issuing for your users.

The `getUserId` function is a helper function that you'll call in resolvers which require authentication (such as `post`). It first retrieves the `Authorization` header (which contains the `User`'s JWT) from the `context`. It then verifies the JWT and retrieves the `User`'s ID from it. Notice that if that process is not successful for any reason, the

To make everything work, be sure to add the following import statements to the top of `Mutation.js` :

.../hackernews-node/src/resolvers/Mutation.js

```
const bcrypt = require('bcryptjs')
const jwt = require('jsonwebtoken')
const { APP_SECRET, getUserId } = require('../utils')
```

Right now, there's one more minor issue. You're accessing a `request` object on the `context` . However, when initializing the `context` , you're really only attaching the `prisma` instance to it - there's no `request` object yet that could be accessed.

To make the above operations possible, open `index.js` and adjust the instantiation of the `GraphQLServer` as follows:

.../hackernews-node/src/index.js

```
const server = new GraphQLServer({
  typeDefs: './src/schema.graphql',
  resolvers,
  context: request => {
    return {
      ...request,
      prisma,
    }
  },
})
```

Instead of attaching an object directly, you're now creating the `context` as a function which *returns* the `context` . The advantage of this approach is that you can attach the HTTP request that carries the incoming GraphQL query (or mutation) to the `context` as well. This will allow your resolvers to read the `Authorization` header and validate if the user who submitted the request is eligible to perform the requested operation.

schema/resolver setup. Right now the `post` resolver is still missing.

In `Mutation.js`, add the following resolver implementation for `post`:

.../hackernews-node/src/resolvers/Mutation.js

```
function post(parent, args, context, info) {
  const userId = getUserId(context)

  return context.prisma.link.create({
    data: {
      url: args.url,
      description: args.description,
      postedBy: { connect: { id: userId } },
    }
  })
}
```

Two things have changed in the implementation compared to the previous implementation in `index.js`:

1. You're now using the `getUserId` function to retrieve the ID of the `User`. This ID is stored in the JWT that's set at the `Authorization` header of the incoming HTTP request. Therefore, you know which `User` is creating the `Link` here. Recall that an unsuccessful retrieval of the `userId` will lead to an exception and the function scope is exited before the `createLink` mutation is invoked. In that case, the GraphQL response will just contain an error indicating that the user was not authenticated.
2. You're then also using that `userId` to *connect* the `Link` to be created with the `User` who is creating it. This is happening through a nested write.

Resolving relations

There's one more thing you need to do before you can launch the GraphQL server again and test the new functionality: ensuring the relation between `User` and `Link` gets properly resolved.

Notice how we've omitted all resolvers for *scalar* values from the `User` and `Link` types? These are following the simple pattern that we saw at the beginning of the

```
    id: parent => parent.id,
    url: parent => parent.url,
    description: parent => parent.description,
  }
```

However, we've now added two fields to our GraphQL schema that can *not* be resolved in the same way: `postedBy` on `Link` and `links` on `User`. The resolvers for these fields need to be explicitly implemented because our GraphQL server can not infer where to get that data from.

To resolve the `postedBy` relation, open `Link.js` and add the following code to it:

.../hackernews-node/src/resolvers/Link.js

```
function postedBy(parent, args, context) {
  return context.prisma.link.findOne({ where: { id: parent.id } }).postedBy()
}

module.exports = {
  postedBy,
}
```

In the `postedBy` resolver, you're first fetching the `Link` from the database using the `prisma` instance and then invoke `postedBy` on it. Notice that the resolver needs to be called `postedBy` because it resolves the `postedBy` field from the `Link` type in `schema.graphql`.

You can resolve the `links` relation in a similar way.

Open `User.js` and add the following code to it:

.../hackernews-node/src/resolvers/User.js

```
function links(parent, args, context) {
  return context.prisma.user.findOne({ where: { id: parent.id } }).links()
}
```

```
}
```

Putting it all together

Awesome! The last thing you need to do now is use the new resolver implementations in `index.js`.

Open `index.js` and import the modules which now contain the resolvers at the top of the file:

.../hackernews-node/src/index.js

```
const Query = require('./resolvers/Query')
const Mutation = require('./resolvers/Mutation')
const User = require('./resolvers/User')
const Link = require('./resolvers/Link')
```

Then, update the definition of the `resolvers` object to looks as follows:

.../hackernews-node/src/index.js

```
const resolvers = {
  Query,
  Mutation,
  User,
  Link
}
```

That's it, you're ready to test the authentication flow! 🔒

Testing the authentication flow

If you haven't done so already, stop and restart the server by first killing it with **CTRL+C**, then run `node src/index.js`. Afterwards, navigate to `http://localhost:4000` where the GraphQL Playground is running.

Note that you can "reuse" your Playground from before if you still have it open - it's only important that you restart the server so the changes you made to the implementation are actually applied.

Now, send the following mutation to create a new `User`:

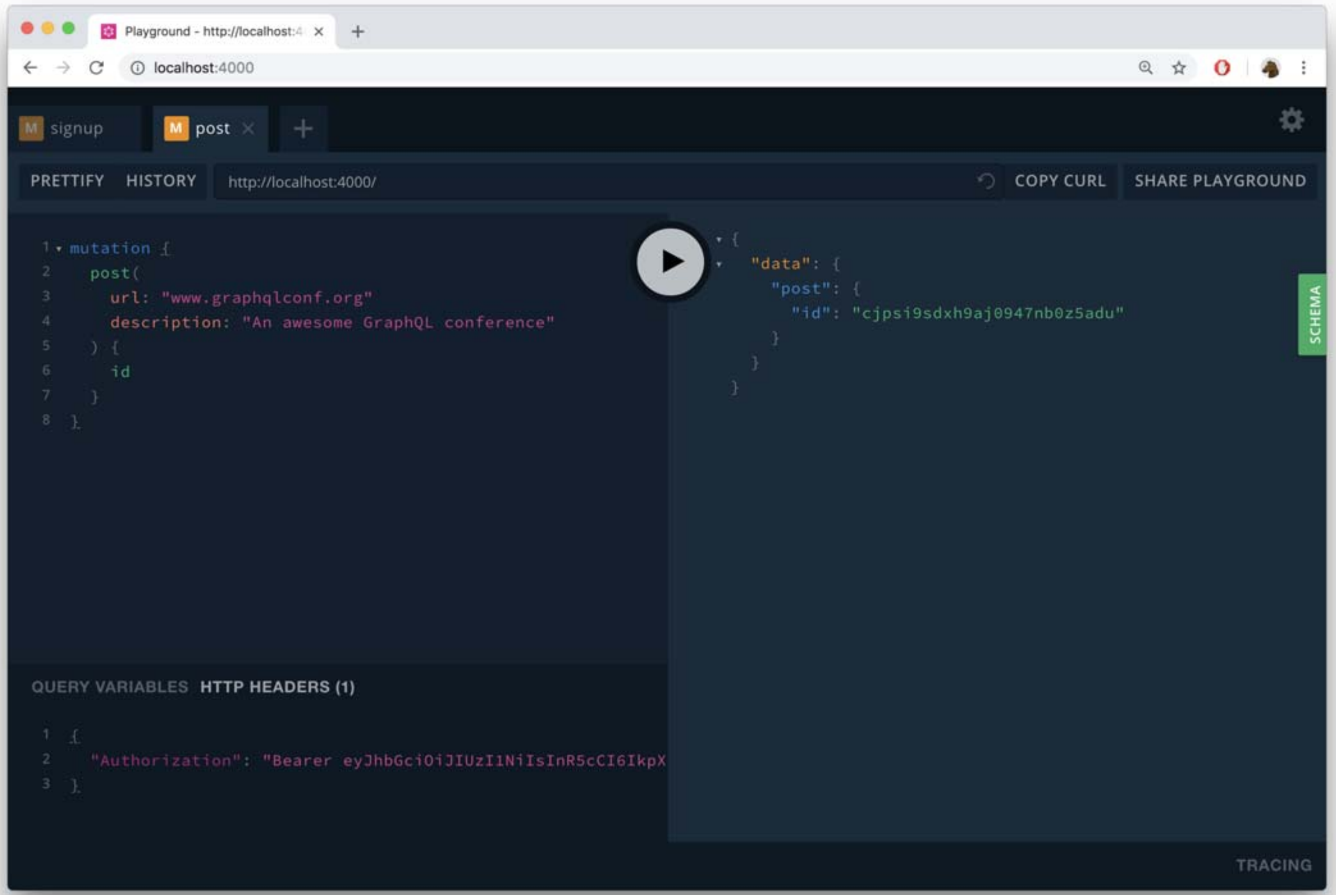
```
mutation {
  signup(
    name: "Alice"
    email: "alice@prisma.io"
    password: "graphql"
  ) {
    token
    user {
      id
    }
  }
}
```

From the server's response, copy the authentication `token` and open another tab in the Playground. Inside that new tab, open the **HTTP HEADERS** pane in the bottom-left corner and specify the `Authorization` header - similar to what you did with the Prisma Playground before. Replace the `__TOKEN__` placeholder in the following snippet with the copied token:

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

With the `Authorization` header in place, send the following to your GraphQL server:

```
mutation {
  post(
    url: "www.graphqlconf.org"
    description: "An awesome GraphQL conference"
  ) {
    id
  }
}
```



```
mutation {
  post(
    url: "www.graphqlconf.org"
    description: "An awesome GraphQL conference"
  ) {
    id
  }
}
```

```
{
  "data": {
    "post": {
      "id": "cjpsi9sdxh9aj0947nb0z5adu"
    }
  }
}
```

```
{
  "Authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX
}
```

When your server receives this mutation, it invokes the `post` resolver and therefore validates the provided JWT. Additionally, the new `Link` that was created is now connected to the `User` for which you previously sent the `signup` mutation.

```
  login(
    email: "alice@prisma.io"
    password: "graphql"
  ) {
    token
    user {
      email
      links {
        url
        description
      }
    }
  }
}
```

This will return a response similar to this:

```
{
  "data": {
    "login": {
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJjanBzaHVsazJ
      "user": {
        "email": "alice@prisma.io",
        "links": [
          {
            "url": "www.graphqlconf.org",
            "description": "An awesome GraphQL conference"
          }
        ]
      }
    }
  }
}
```

UNLOCK THE NEXT CHAPTER

Which HTTP header field carries the authentication token?

○ Authorization

○ Authentication

Skip

Edit on Github