WRITTEN BY

Robin MacPherson

Software Engineer @ StyleSeat // CEO @ Journaly

Robin is a full-stack software engineer and budding data scientist who also loves learning foreign (human) languages. He is a co-founder @ Journaly.io, runs a language learning YouTube channel, and works at StyleSeat in San Francisco.

Getting Started

In this section, you will set up the project for your GraphQL server and implement your first GraphQL query. At the end, we'll talk theory for a bit and learn about the GraphQL schema.

Creating the project

This tutorial teaches you how to build a GraphQL server from scratch, so the first thing you need to do is create the directory that'll hold the files for your GraphQL server!

- Open your terminal, navigate to a location of your choice, and run the following commands:

```
mkdir hackernews-node
cd hackernews-node
npm init -y
```

This creates a new directory called `hackernews-node` and initializes it with a `package.json` file. `package.json` is the configuration file for the Node.js app you're building. It lists all dependencies and other configuration options (such as *scripts*) needed for the app.

Creating a raw GraphQL server

With the project directory in place, you can go ahead and create the entry point for your GraphQL server. This will be a file called `index.js`, located inside a directory called `src`.

- In your terminal, first create the `src` directory and then the empty `index.js` file:

$ .../hackernews-node/

```
mkdir src
touch src/index.js
```

> **Note**: The above code block is annotated with a directory name. It indicates *where* you need to execute the terminal command.

To start the app, you can now execute `node src/index.js` inside the `hackernews-node` directory. At the moment, this won't do anything because `index.js` is still empty ¯_(ツ)_/¯

Let's go and start building the GraphQL server! The first thing you need to is - surprise - add a dependency to the project.

First, let's install an important dependency that will allow you to create your GraphQL server.

- Run the following command in your terminal:

$ .../hackernews-node/

```
npm install apollo-server
```

`apollo-server` is a fully-featured GraphQL server. It is based on Express.js and a few other libraries to help you build production-ready GraphQL servers.

Here's a list of its features:

- GraphQL spec-compliant
- Realtime functionality with GraphQL subscriptions
- Out-of-the-box support for GraphQL Playground
- Extensible via Express middlewares
- Resolves custom directives in your GraphQL schema
- Query performance tracing
- Runs everywhere: Can be deployed via Vercel, Up, AWS Lambda, Heroku etc.

Perfect, it's time to write some code 🙌

- Open `src/index.js` and type the following:

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

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

// 1
const typeDefs = `
  type Query {
    info: String!
  }
`

// 2
const resolvers = {
  Query: {
    info: () => `This is the API of a Hackernews Clone`
  }
}
```

```
// 3
const server = new ApolloServer({
  typeDefs,
  resolvers,
})

server
  .listen()
  .then(({ url }) =>
    console.log(`Server is running on ${url}`)
  );
```

who calls index.js ?

> **Note**: This code block is annotated with a file name. It indicates into which file you need to put the code that's shown. The annotation also links to the corresponding file on GitHub to help you figure out *where* in the file you need to put it in case you are not sure about that.

All right, let's understand what's going on here by walking through the numbered comments:

1. The `typeDefs` constant defines your *GraphQL schema* (more about this in a bit). Here, it defines a simple `Query` type with one *field* called `info`. This field has the type `String!`. The exclamation mark in the type definition means that this field is required and can never be `null`.
2. The `resolvers` object is the actual *implementation* of the GraphQL schema. Notice how its structure is identical to the structure of the type definition inside `typeDefs`: `Query.info`.
3. Finally, the schema and resolvers are bundled and passed to `ApolloServer` which is imported from `apollo-server`. This tells the server what API operations are accepted and how they should be resolved.

Go ahead and test your GraphQL server!

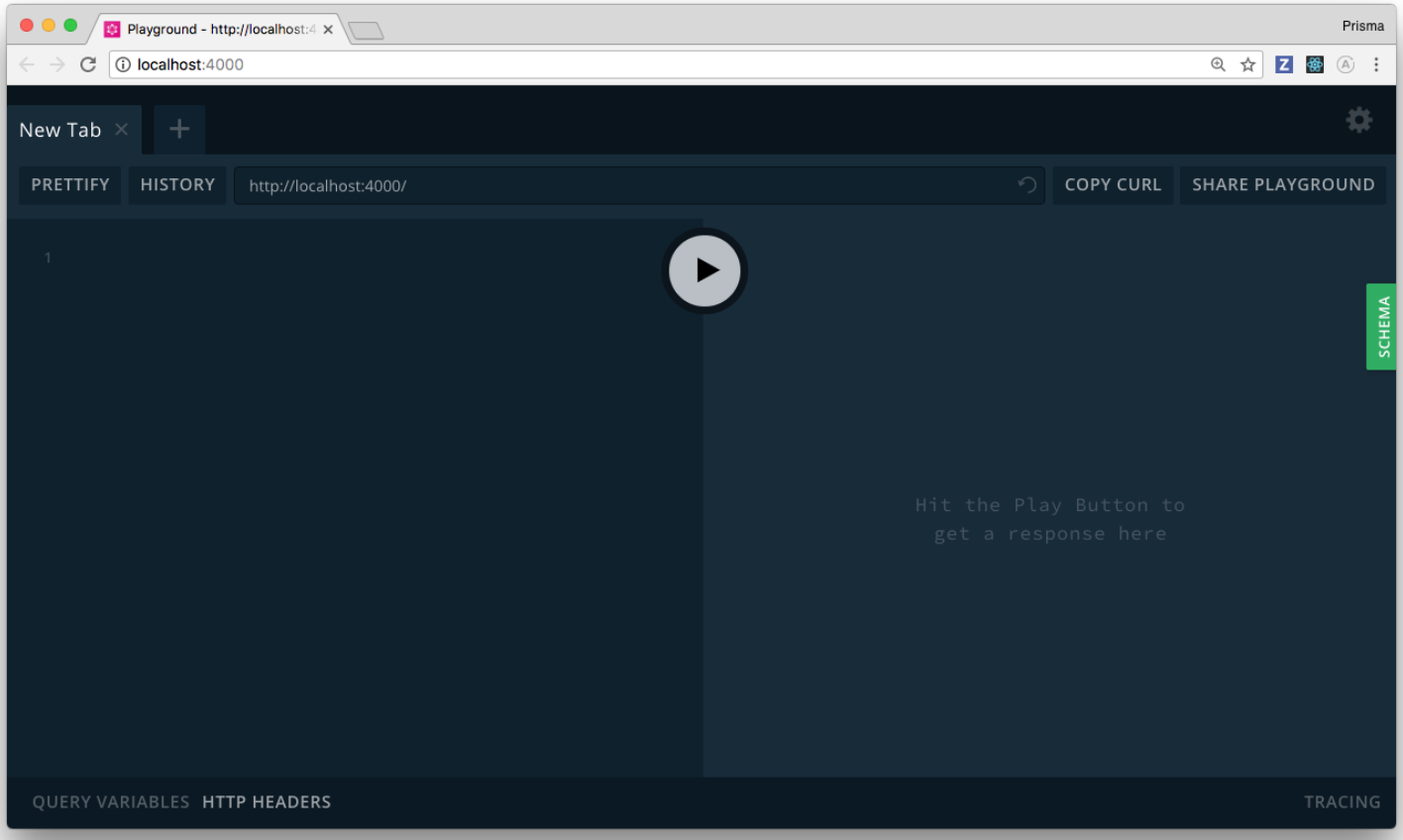
Testing the GraphQL server

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

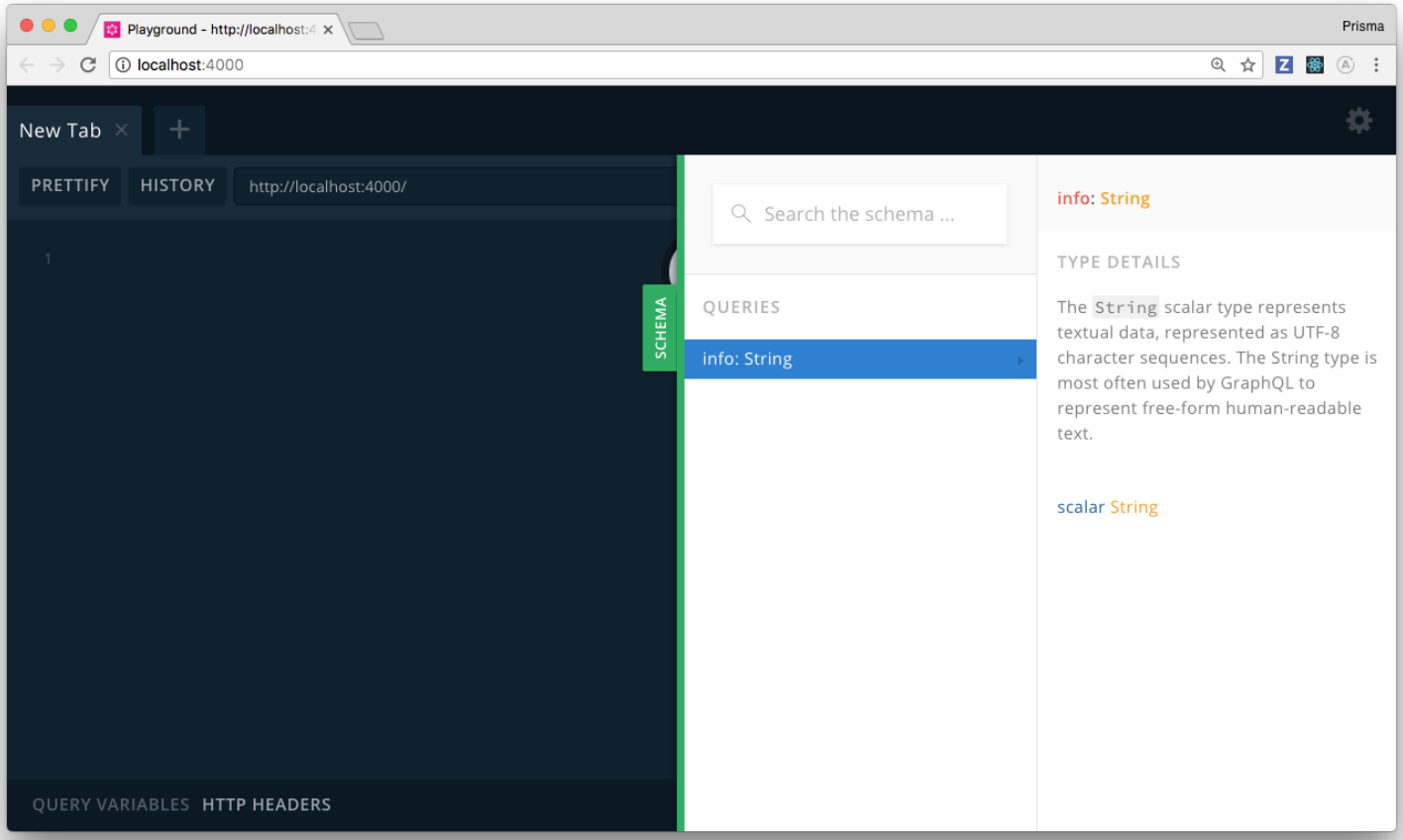
```
$ .../hackernews-node/
node src/index.js
```

As indicated by the terminal output, the server is now running on `http://localhost:4000`. To test the API of your server, open a browser and navigate to that URL.

What you'll then see is a GraphQL Playground, a powerful "GraphQL IDE" that lets you explore the capabilities of your API in an interactive manner.



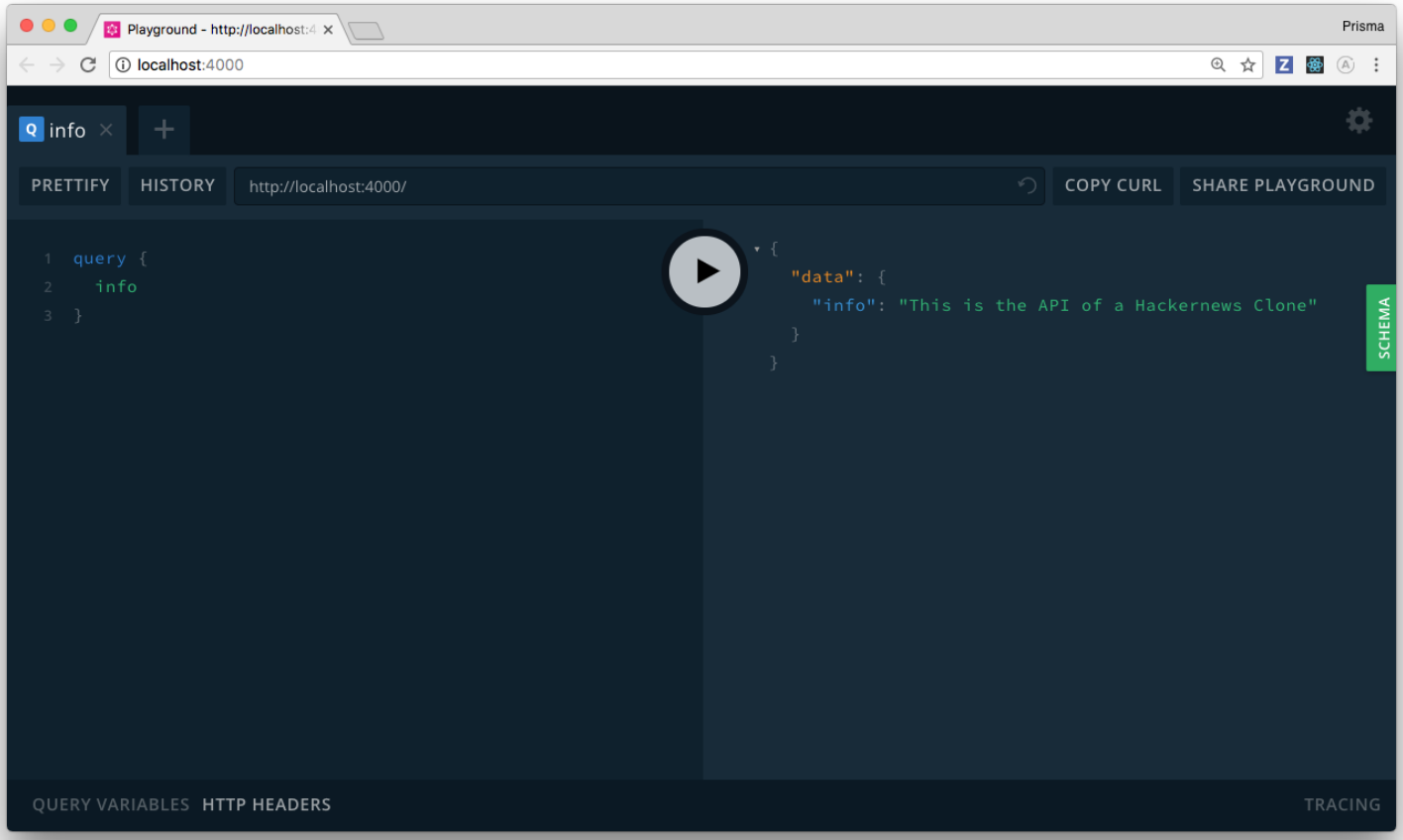
By clicking the **DOCS**-button on the right, you can open the API documentation. This documentation is auto-generated based on your schema definition and displays all API operations and data types of your schema.



Let's go ahead and send your very first GraphQL query. Type the following into the editor pane on the left side:

```
query {
  info
}
```

Now send the query to the server by clicking the **Play**-button in the center (or use the keyboard shortcut **CMD+ENTER** for Mac and **CTRL+ENTER** on Windows and Linux).



Congratulations, you just implemented and successfully tested your first GraphQL query 🎉

Now, remember when we talked about the definition of the `info: String!` field and said the exclamation mark means this field could never be `null`. Well, since you're implementing the resolver, you are in control of what the value for that field is, right?

So, what happens if you return `null` instead of the actual informative string in the resolver implementation? Feel free to try that out!

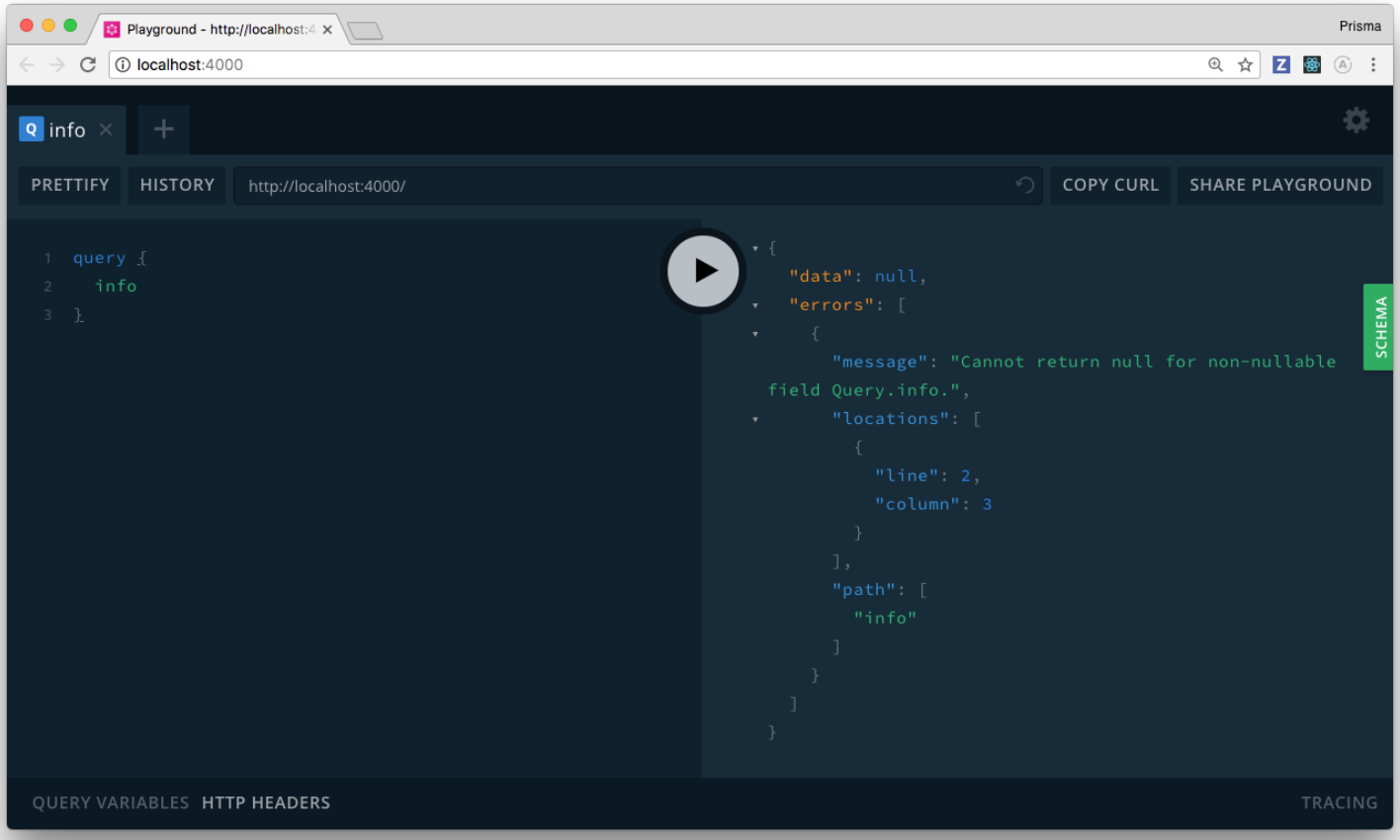
In `index.js`, update the the definition of `resolvers` as follows:

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

```
const resolvers = {
  Query: {
    info: () => null,
  }
}
```

To test the results of this, you need to restart the server: First, stop it using **CTRL+C** on your keyboard, then restart it by running `node src/index.js` again.

Now, send the query from before again. This time, it returns an error:
`Error: Cannot return null for non-nullable field Query.info.`



(schema) (schema 작성)
typeDef > resolver

What happens here is that the underlying `graphql-js` reference implementation ensures that the return types of your resolvers adhere to the type definitions in your GraphQL schema. Put differently, it protects you from making stupid mistakes!

This is in fact one of the core benefits of GraphQL in general: it enforces that the API actually behaves in the way that is promised by the schema definition! This way, everyone who has access to the GraphQL schema can always be 100% sure about the API operations and data structures that are returned by the API.

A word on the GraphQL schema

At the core of every GraphQL API, there is a GraphQL schema. So, let's quickly talk about it.

> **Note**: In this tutorial, we'll only scratch the surface of this topic. If you want to go a bit more in-depth and learn more about the GraphQL schema as well as its role in a GraphQL API, be sure to check out this excellent article.

GraphQL schemas are usually written in the GraphQL Schema Definition Language (SDL). SDL has a type system that allows you to define data structures (just like other strongly typed programming languages such as Java, TypeScript, Swift, Go, etc.).

How does that help in defining the API for a GraphQL server, though? Every GraphQL schema has three special *root types*: `Query`, `Mutation`, and `Subscription`. The root types correspond to the three operation types offered by GraphQL: queries, mutations, and subscriptions. The fields on these root types are called *root fields* and define the available API operations.

As an example, consider the simple GraphQL schema we used above:

```
type Query {
  info: String!
}
```

This schema only has a single root field, called `info`. When sending queries, mutations or subscriptions to a GraphQL API, these always need to start with a root field! In this case, we only have one root field, so there's really only one possible query that's accepted by the API.

Let's now consider a slightly more advanced example:

```
type Query {
  users: [User!]!
  user(id: ID!): User
}

type Mutation {
  createUser(name: String!): User!
}

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

In this case, we have three root fields: `users` and `user` on `Query` as well as `createUser` on `Mutation`. The additional definition of the `User` type is required because otherwise the schema definition would be incomplete.

What are the API operations that can be derived from this schema definition? Well, we know that each API operation always needs to start with a root field. However, we haven't learned yet what it looks like when the *type* of a root field is itself another object type. This is the case here, where the types of the root fields are `[User!]!`, `User` and `User!`. In the `info` example from before, the type of the root field was a `String`, which is a scalar type.

When the type of a root field is an object type, you can further expand the query (or mutation/subscription) with fields of that object type. The expanded part is called *selection set*.

Here are the operations that are accepted by a GraphQL API that implements the above schema:

```
# Query for all users
query {
  users {
    id
    name
  }
}

# Query a single user by their id
query {
  user(id: "user-1") {
    id
    name
  }
}

# Create a new user
mutation {
  createUser(name: "Bob") {
    id
    name
  }
}
```

There are a few things to note:

- In these examples, we always query `id` and `name` of the returned `User` objects. We could potentially omit either of them. Note, however, when querying an object type, it is required that you query at least one of its fields in a selection set.
- For the fields in the selection set, it doesn't matter whether the type of the root field is *required* or a *list*. In the example schema above, the three root fields all have different type modifiers (i.e. different combinations of being a list and/or required) for the `User` type:
  - For the `users` field, the return type `[User!]!` means it returns a *list* (which itself cannot be `null`) of `User` elements. The list can also not contain elements that are `null`. So, you're always guaranteed to either receive an empty list or a list that only contains non-null `User` objects.
  - For the `user(id: ID!)` field, the return type `User` means the returned value could be `null` *or* a `User` object.
  - For the `createUser(name: String!)` field, the return type `User!` means this operation always returns a `User` object.

Phew, enough theory 😠 Let's go and write some more code!

UNLOCK THE NEXT CHAPTER

What role do the root fields play for a GraphQL API?

- [x] The three root fields are: Query, Mutation and Subscription
- [ ] Root fields implement the available API operations
- [ ] Root fields define the available API operations
- [ ] Root field is another term for resolver

Skip

Edit on Github

A Simple Query

In this section, you are going to implement the first API operation that provides the functionality of a Hacker News clone: querying a feed of *links* that were posted by other users.

Extending the schema definition

Let's start by implementing a `feed` query which allows you to retrieve a list of `Link` elements. In general, when adding a new feature to the API, the process will look pretty similar every time:

1. Extend the GraphQL schema definition with a new *root field* (and new *object types*, if needed)
2. Implement corresponding *resolver functions* for the added fields

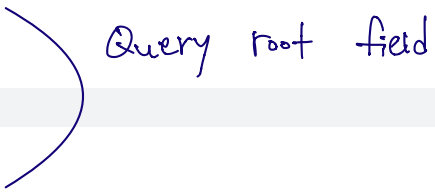
This process is also referred to as *schema-driven* or *schema-first development*.

So, let's go ahead and tackle the first step, extending the GraphQL schema definition.

- In `index.js`, update the `typeDefs` constant to look as follows:

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

```
const typeDefs = `
  type Query {
    info: String!
    feed: [Link!]!
  }

  type Link {
    id: ID!
    description: String!
    url: String!
```



```
  }
`
```

Pretty straightforward, right? You're defining a new `Link` type that represents the links that can be posted to Hacker News. Each `Link` has an `id`, a `description`, and a `url`. You're then adding another root field to the `Query` type that allows you to retrieve a list of `Link` elements. This list is guaranteed to never be `null` (if anything, it will be empty) and never contain any elements that are `null` - that's what the two exclamation marks are for.

Implement resolver functions

The next step is to implement the resolver function for the `feed` query. In fact, one thing we haven't mentioned yet is that not only *root fields*, but virtually *all* fields on the types in a GraphQL schema have resolver functions. So, you'll add resolvers for the `id`, `description`, and `url` fields of the `Link` type as well.

- In `index.js`, add a new list with dummy data as well and update the `resolvers` to look as follows:

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

```
// 1
let links = [{
  id: 'link-0',
  url: 'www.howtographql.com',
  description: 'Fullstack tutorial for GraphQL'
}]

const resolvers = {
  Query: {
    info: () => `This is the API of a Hackernews Clone`,
    // 2
    feed: () => links,
  },
  // 3
  Link: {
    id: (parent) => parent.id,
    description: (parent) => parent.description,
    url: (parent) => parent.url,
```

① feed resolver invoked

↳ this is a "feed" resolver

② schema의 feed: [Link!]! 이므로 resolver의 Link resolver 함수 parent 에는 Link 타입이 들어간다.

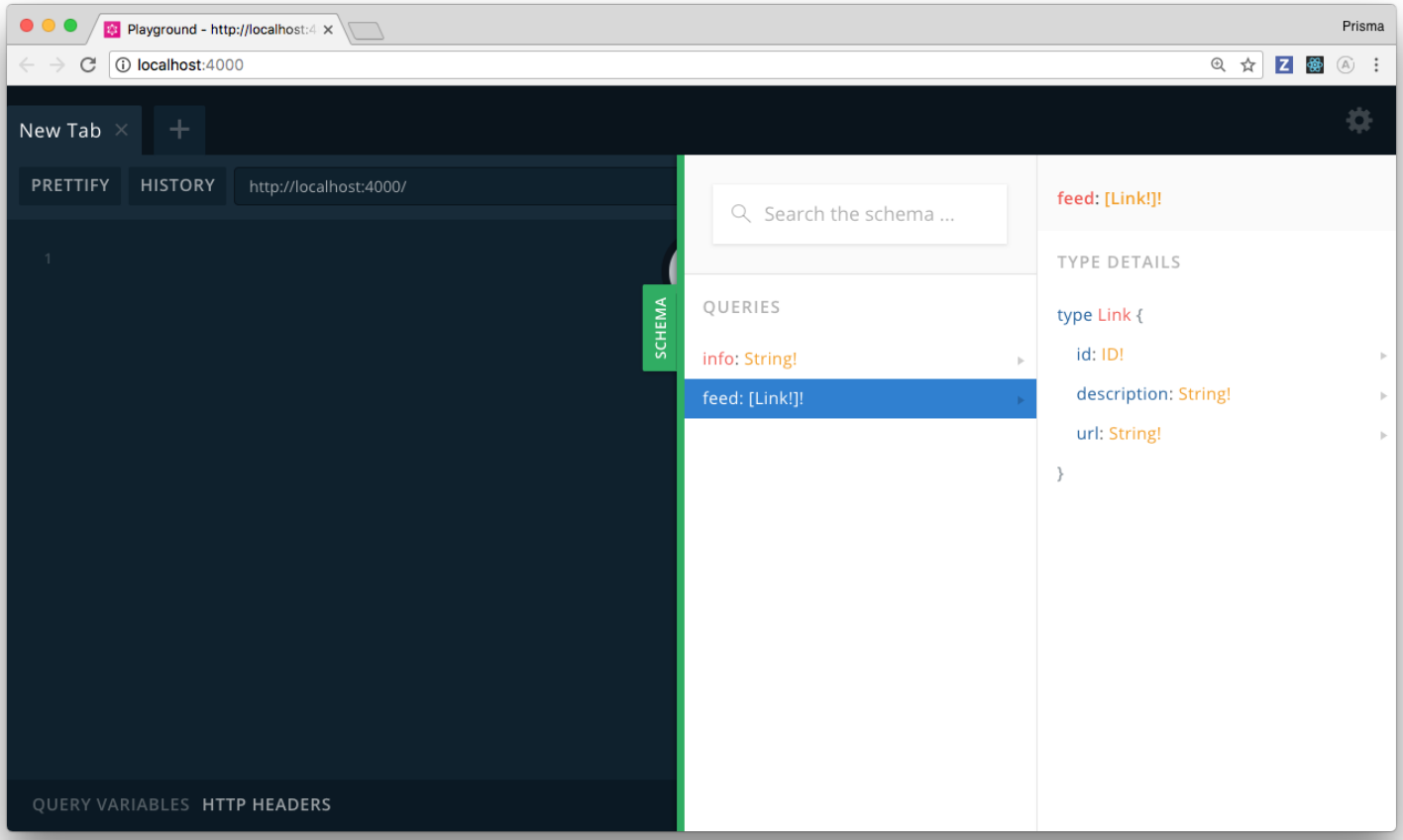
schema (typedef) 에 정의된 type

```
  }
}
```

Let's walk through the numbered comments again:

1. The `links` variable is used to store the links at runtime. For now, everything is stored only *in-memory* rather than being persisted in a database.
2. You're adding a new resolver for the `feed` root field. Notice that a resolver always has to be named *exactly* after the corresponding field from the schema definition.
3. Finally, you're adding three more resolvers for the fields on the `Link` type from the schema definition. We'll discuss what the `parent` argument that's passed into the resolver here is in a bit.

Go ahead and test the implementation by restarting the server (first use **CTRL+C** to stop the server if it is still running, then execute `node src/index.js` again) and navigate to `http://localhost:4000` in your browser. If you expand the documentation of the Playground, you'll notice that another query called `feed` is now available:



Try it out by sending the following query:

```
query {
  feed {
    id
    url
    description
  }
}
```

Awesome, the server responds with the data you defined in `links`:

```
{
  "data": {
    "feed": [
      {
        "id": "link-0",
        "url": "www.howtographql.com",
        "description": "Fullstack tutorial for GraphQL"
      }
```

```
    ]
  }
}
```

Feel free to play around with the query by removing any fields from the selection set and observe the responses sent by the server.

The query resolution process

Let's now quickly talk about how a GraphQL server actually resolves incoming queries. As you already saw, a GraphQL query consists of a number of *fields* that have their source in the type definitions of the GraphQL schema.

Let's consider the query from above again:

```
query {
  feed {
    id
    url
    description
  }
}
```

All four fields specified in the query (`feed`, `id`, `url`, and `description`) can also be found inside the schema definition. Now, you also learned that *every* field inside the schema definition is backed by one resolver function whose responsibility it is to return the data for precisely that field.

S) field에 맞는 resolver들을 invoke

Can you imagine what the query resolution process looks like now? Effectively, all the GraphQL server has to do is invoke all resolver functions for the fields that are contained in the query and then package up the response according to the query's shape. Query resolution thus merely becomes a process of orchestrating the invocation of resolver functions!

GraphQL 서버의 역할

One thing that's still a bit weird in the implementation right now are the resolvers for the `Link` type that all seem to follow a very simple and trivial pattern:

```
Link: {
  id: (parent) => parent.id,
  description: (parent) => parent.description,
```

```
  url: (parent) => parent.url,
}
```

First, it's important to note that every GraphQL resolver function actually receives *four* input arguments. As the remaining three are not needed in our scenario right now, we're simply omitting them. Don't worry, you'll get to know them soon.

The first argument, commonly called `parent` (or sometimes `root`) is the result of the previous *resolver execution level*. Hang on, what does that mean? 🤔

Well, as you already saw, GraphQL queries can be *nested*. Each level of nesting (i.e. nested curly braces) corresponds to one resolver execution level. The above query therefore has two of these execution levels.

On the first level, it invokes the `feed` resolver and returns the entire data stored in `links`. For the second execution level, the GraphQL server is smart enough to invoke the resolvers of the `Link` type (because thanks to the schema, it knows that `feed` returns a list of `Link` elements) for each element inside the list that was returned on the previous resolver level. Therefore, in all of the three `Link` resolvers, the incoming `parent` object is the element inside the `links` list.

> **Note**: To learn more about this, check out this article.

In any case, because the implementation of the `Link` resolvers is trivial, you can actually omit them and the server will work in the same way as it did before 👌 We just wanted you to understand what's happening under the hood 🚗

UNLOCK THE NEXT CHAPTER

How are GraphQL queries resolved?

- ◯ With schema-driven development
- ◯ By invoking all available resolver functions

- By invoking the resolver function of the root field
- By invoking the resolver functions for the fields contained in the query

Skip

Edit on Github

A Simple Mutation

In this section, you'll learn how to add a mutation to the GraphQL API. This mutation will allow you to *post* new links to the server.

Extending the schema definition

Like before, you need to start by adding the new operation to your GraphQL schema definition.

In `index.js`, extend the `typeDefs` string as follows:

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

```
const typeDefs = `
  type Query {
    info: String!
    feed: [Link!]!
  }

  type Mutation {
    post(url: String!, description: String!): Link!
  }

  type Link {
    id: ID!
    description: String!
    url: String!
  }
`
```

PlayGround 에서 ref로 동작할까?

At this point, the schema definition has already grown to be quite large. Let's refactor

the app a bit and pull the schema out into its own file!

- Create a new file inside the `src` directory and call it `schema.graphql`:

```
$ ../hackernews-node/src
```

```
touch src/schema.graphql
```

- Next, copy the entire schema definition into the new file:

../hackernews-node/src/schema.graphql

```
type Query {
  info: String!
  feed: [Link!]!
}

type Mutation {
  post(url: String!, description: String!): Link!
}

type Link {
  id: ID!
  description: String!
  url: String!
}
```

With that new file in place, you can cleanup `index.js` a bit.

- First, entirely delete the definition of the `typeDefs` constant - it's not needed any more because the schema definition now lives in its own file. Then, update the way that the `GraphQLServer` is instantiated at the bottom of the file:

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

```
const fs = require('fs');
const path = require('path');

const server = new ApolloServer({
  typeDefs: fs.readFileSync(
    path.join(__dirname, 'schema.graphql'),
    'utf8'
  ),
  resolvers,
})
```

why Sync?

One convenient thing about the constructor of the `GraphQLServer` is that `typeDefs` can be provided either directly as a string (as you previously did) or by referencing a file that contains your schema definition (this is what you're doing now).

① Schema를 ``` string으로 전달
② .graphql 파일 import 전달 (text도 가능함)

Implementing the resolver function

The next step in the process of adding a new feature to the API is to implement the resolver function for the new field.

- Next, update the `resolvers` functions to look as follows:

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

```
let links = [{
  id: 'link-0',
  url: 'www.howtographql.com',
  description: 'Fullstack tutorial for GraphQL'
}]

// 1
let idCount = links.length
const resolvers = {
  Query: {
    info: () => `This is the API of a Hackernews Clone`,
    feed: () => links,
  },
  Mutation: {
    // 2
    post: (parent, args) => {
       const link = {
```

```
          id: `link-${idCount++}`,
          description: args.description,
          url: args.url,
        }
        links.push(link)
        return link
      }
    },
  }
```

First off, note that you're entirely removing the `Link` resolvers (as explained at the very end of the last section). They are not needed because the GraphQL server infers what they look like.

Also, here's what's going on with the numbered comments:

1. You're adding a new integer variable that simply serves as a very rudimentary way to generate unique IDs for newly created `Link` elements.
2. The implementation of the `post` resolver first creates a new `link` object, then adds it to the existing `links` list and finally returns the new `link`.

Now's a good time to discuss the second argument that's passed into all resolver functions: `args`. Any guesses what it's used for?

Correct! 💡 It carries the *arguments* for the operation – in this case the `url` and `description` of the `Link` to be created. We didn't need it for the `feed` and `info` resolvers before, because the corresponding root fields don't specify any arguments in the schema definition.

Testing the mutation

Go ahead and restart your server so you can test the new API operations. Here is a sample mutation you can send through the GraphQL Playground:

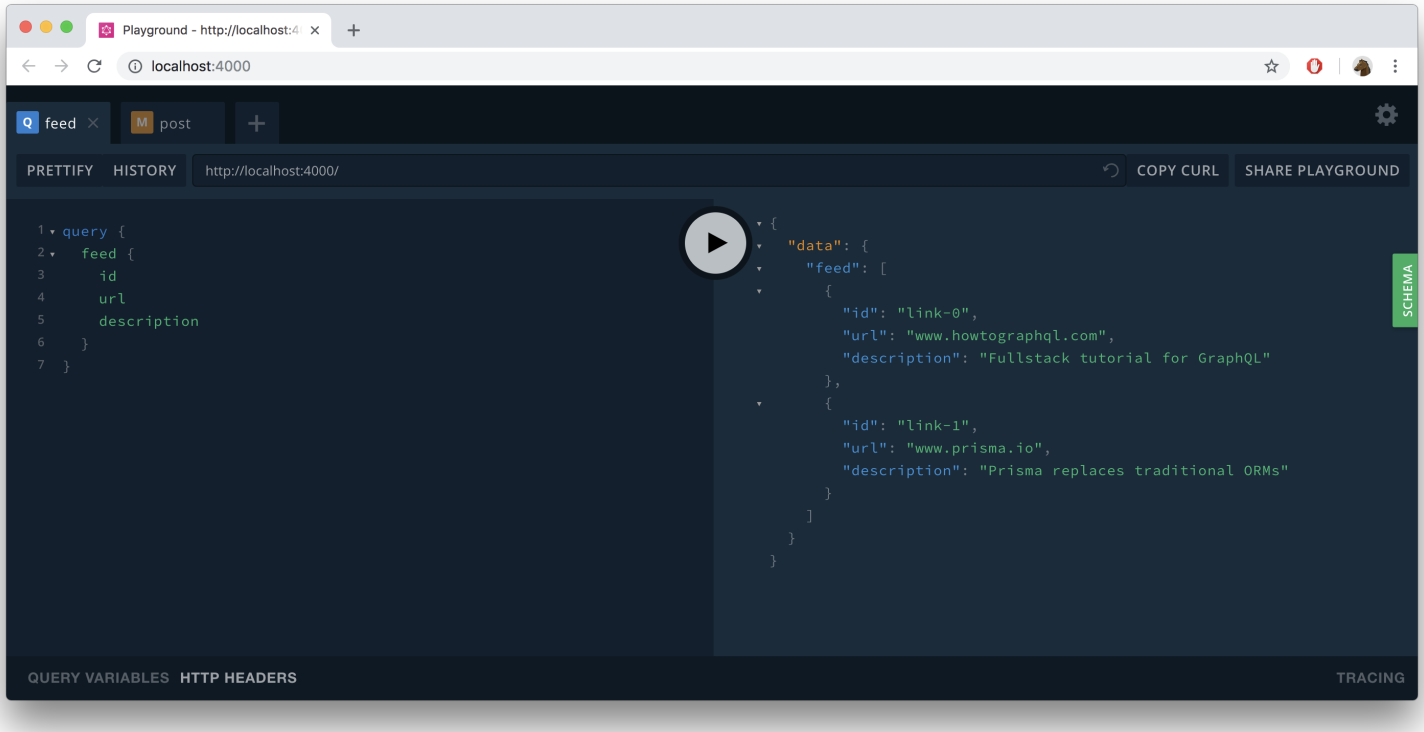
```
mutation {
  post(url: "www.prisma.io", description: "Prisma replaces traditional ORMs") {
    id
  }
}
```

The server response will look as follows:

```
{
  "data": {
    "post": {
      "id": "link-1"
    }
  }
}
```

With every mutation you send, the `idCount` will increase and the following IDs for created links will be `link-2`, `link-3`, and so forth...

To verify that your mutation worked, you can send the `feed` query from before again – it now returns the additional Link that you created with the mutation:



However, once you kill and restart the server, you'll notice that the previously added links are now gone and you need to add them again. This is because the links are only stored *in-memory*, in the `links` array. In the next sections, you will learn how to add a *database* to the GraphQL server in order to persist the data beyond the runtime of the server.

Exercise

If you want to practice implementing GraphQL resolvers a bit more, here's an *optional* challenge for you. Based on your current implementation, extend the GraphQL API with full CRUD functionality for the `Link` type. In particular, implement the queries and mutations that have the following definitions:

```
type Query {
  # Fetch a single link by its `id`
  link(id: ID!): Link
}

type Mutation {
  # Update a link
  updateLink(id: ID!, url: String, description: String): Link

  # Delete a link
  deleteLink(id: ID!): Link
}
```

UNLOCK THE NEXT CHAPTER

What is the second argument that's passed into GraphQL resolvers used for?

- ○ It carries the return value of the previous resolver execution level
- ✓ It carries the arguments for the incoming GraphQL operation
- ○ It is an object that all resolvers can write to and read from
- ○ It carries the AST of the incoming GraphQL operation

Skip

Edit on Github

Adding a Database

In this section, you're going to set up a SQLite to persist the data of incoming GraphQL mutations. Instead of writing SQL directly, you will use Prisma to access your database.

So, what is Prisma?

Prisma is an open source database toolkit that makes it easy for developers to reason about their data and how they access it, by providing a clean and type-safe API for submitting database queries.

It mainly consists of three tools:

- **Prisma Client**: An auto-generated and type-safe query builder for Node.js & TypeScript. (Query를 만들어줌)
- **Prisma Migrate** (experimental): A declarative data modeling & migration system.
- **Prisma Studio** (experimental): A GUI to view and edit data in your database.

In this tutorial, you will be setting everything up from scratch and taking full advantage of these three tools. We want to get you building stuff right away, so explanations of Prisma concepts will be kept light but we have included links to Prisma docs in case you want to dive deeper on any particular concept.

Why Prisma?

You've now understood the basic mechanics of how GraphQL servers work under the hood and the beauty of GraphQL itself – it actually only follows a few very simple rules. The statically typed schema and the GraphQL engine that resolves the queries

이 엔진이 SDL을 translate 해주는거야.

inside the server take away major pain points commonly dealt with in API development.

Well, in real-world applications you're likely to encounter many scenarios where implementing the resolvers can become extremely complex. Especially because GraphQL queries can be nested multiple levels deep, the implementation often becomes tricky and can easily lead to performance problems.

즉 prisma를 사용하면 performance issue를 잡는다? (mongoDB 와 조합시 얘는 어떻게 작동?)

Most of the time, you also need to take care of many additional workflows such as authentication, authorization (permissions), pagination, filtering, realtime, integrating with 3rd-party services or legacy systems, and so on.

Prisma is focused on addressing that issue and making developers more productive when working with databases.

Speaking of being productive and building awesome stuff, let's jump back in and continue with our HackerNews Clone! 🏎️💨

Setting up our project with Prisma and SQLite

- First, let's install the Prisma CLI by running the following command in your terminal:

```
$ .../hackernews-node/
npm install @prisma/cli --save-dev
```

Next, use the Prisma CLI to initialize Prisma in the project.

- From your project root, run the following commands in your terminal:

```
$ .../hackernews-node/
npx prisma init
```

Remember the GraphQL schema that you've been working with until now? Well, Prisma has a schema, too! Inside the `prisma` directory that was created in the last step, you'll see a file called `schema.prisma`. You can think of the `prisma.schema` file as a *database schema*. It has three components:

1. **Data source**: Specifies your database connection.
2. **Generator**: Indicates that you want to genenerate Prisma Client.
3. **Data model**: Defines your application *models*. Each model will be mapped to a table in the underlying database.

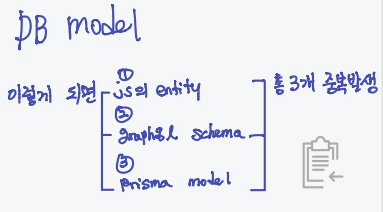
Prisma's unique data model bridges the gap to help you reason about your data in a way that maps very well to the underlying database, while still providing an abstraction that allows you to be productive with type safety and auto-completion.

Let's see it in action with your project!

Open `schema.prisma` and add the following code:

.../hackernews-node/prisma/schema.prisma

```
// 1
datasource db {
  provider = "sqlite"
  url      = "file:./dev.db"
}

// 2
generator client {
  provider = "prisma-client-js"
}

// 3
model Link {
  id          Int      @id @default(autoincrement())
  createdAt   DateTime @default(now())
  description String
  url         String
}
```



Let's break down the three parts:

1. **Data source**: Tells Prisma you'll be using SQLite for your database connection.
2. **Generator**: Indicates that you want to genenerate Prisma Client.
3. **Data model**: Here, we have written out our `Link` as a model.

The `Link` model defines the structure of the `Link` database table that Prisma is going to create for you in a bit.

Getting Started with SQLite

It's finally time to actually create our SQLite database. In case you aren't familiar with SQLite, it is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.

The great thing is that, unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. This makes it a perfect choice for projects like this.

So how about the setup? Well, the great news is that Prisma can do that for us right out of the box with a simple command!

- From the root directory of your project, create your first *migration* by running the following command in your terminal:

```
$ .../hackernews-node/
npx prisma migrate dev --preview-feature
```

- You will get a prompt asking you to provide a name for the migration. Let's name it "init". Type in the name and hit **Enter**.

Take a look at the `prisma` directory in your project's file system. You'll see that there is now a `/migrations` directory that Prisma Migrate created for you when running the above command.

For now, the important thing to understand is that we have told Prisma with our data model, "I want to create a `Link` table to store data about *links*, and here's what that data will look like. Prisma then generates the necessary migration and packages it into a dedicated directory with its own `README.md` file containing detailed information about the specific migration. This is then put inside that `prisma/migrations` directory, which becomes a historical reference of how your database evolves over time with each individual migration you make!

Boom! 💥 You now have a database with a `Link` table! 🥳

Check out the Prisma Migrate docs for a deeper dive on this.

Generating Prisma Client

It's time to generate *Prisma Client* based on your data model!

- Run the following command in your terminal:

```
$ .../hackernews-node/
npx prisma generate
```

It's as simple as that! You now have `/node_modules/@prisma/client` which can be imported and used in your code.

Let's write your first query with Prisma Client and break everything down. You'll do that in a separate file to not mess with your current GraphQL server implementation.

- Create a new file in the `src/` directory called `script.js` and add the following code:

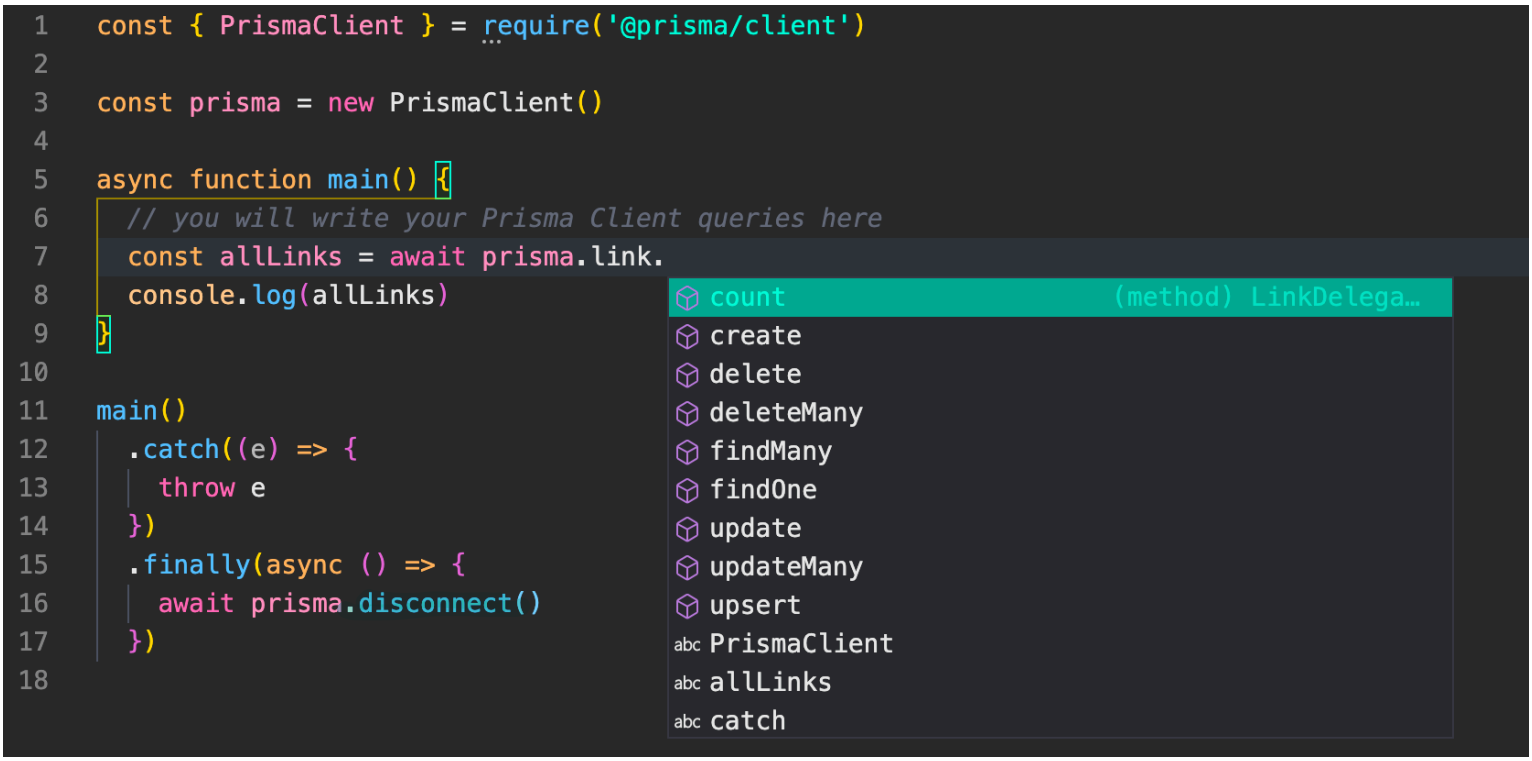
.../hackernews-node/src/script.js

```
// 1
const { PrismaClient } = require("@prisma/client")

// 2
const prisma = new PrismaClient()

//3
async function main() {
  const allLinks = await prisma.link.findMany()
  console.log(allLinks)
}

//4
main()
  .catch(e => {
    throw e
  })
  // 5
  .finally(async () => {
    await prisma.$disconnect()
  })
```

Let's break down what's going on here with the numbered comments:

1. Import the `PrismaClient` constructor from the `@prisma/client` node module.
2. Instantiate `PrismaClient`.
3. Define an `async` function called `main` to send queries to the database. You will write all your queries inside this function.
4. Call the `main` function.
5. Close the database connections when the script terminates.

Take a moment to re-type the query line and notice the helpful autocompletion you get after typing `prisma.` and `prisma.link.` which lets us see all of the possible models we can access and operations we can use to query that data:



So now let's see things in action.

- Run your new code with the following command:

```
$ .../hackernews-node/
node src/script.js
```

You successfully queried the database with Prisma Client! Of course, we got an empty array back since the database is empty, so now let's quickly create a new link in the same script.

- Type out the following lines of code yourself inside of the `main` function right above the `allLinks` query and pay close attention to the incredibly helpful autocompletion. First, it helps us understand that `.create()` is the operation we need (just scroll through the options) and then it actually shows us exactly how to construct the mutation!

.../hackernews-node/src/script.js

```
const newLink = await prisma.link.create({
    data: {
      description: 'Fullstack tutorial for GraphQL',
      url: 'www.howtographql.com',
    },
  })
```

Great! Re-run the previous command and this time you should now see your newly created link print in the terminal output! Much more satisfying ✨

Summary of your workflow

To recap, this is the typical workflow you will follow when updating your data:

1. Manually adjust your Prisma data model. Prisma.schema
2. Migrate your database using the `prisma migrate` CLI commands we covered.
3. (Re-)generate Prisma Client
4. Use Prisma Client in your application code to access your database.

In the next chapters, you will evolve the API of your GraphQL server and use Prisma Client to access the database from inside your resolver functions.

UNLOCK THE NEXT CHAPTER

What is the role of Prisma Client in the GraphQL API?

- [x] It receives and processes GraphQL queries
- [ ] It connects your GraphQL resolvers to the database
- [ ] It migrates your database schema
- [ ] It lets you design your GraphQL schema

Skip

Edit on Github

Connecting The Server and Database with Prisma Client

In this section, you're going to learn how to connect your GraphQL server to your database using Prisma, which provides the interface to your database. This connection is implemented via Prisma Client.

Wiring up your GraphQL schema with Prisma Client

The first thing you need to do is import your generated Prisma Client library and wire up the GraphQL server so that you can access the database queries that your new Prisma Client exposes.

The GraphQL `context` resolver argument

Remember how we said earlier that all GraphQL resolver functions *always* receive four arguments? To accomplish this step, you'll need to get to know another one – the `context` argument!

The `context` argument is a plain JavaScript object that every resolver in the resolver chain can read from and write to. Thus, it is basically a means for resolvers to communicate. A really helpful feature is that you can already write to the `context` at the moment when the GraphQL server itself is being initialized.

This means that we can attach an instance of Prisma Client to the `context` when initializing the server and then access it from inside our resolvers via the `context` argument!

That's all a bit theoretical, so let's see how it looks in action 💻

Updating the resolver functions to use Prisma Client

- First, import `PrismaClient` into `index.js` at the top of the file:

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

```
const { PrismaClient } = require('@prisma/client')
```

Now you can attach an instance of PrismaClient to the `context` when the `GraphQLServer` is being initialized.

- In `index.js`, save an instance of PrismaClient to a variable and update the instantiation of the `GraphQLServer` to add is to the context as follows:

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

```
const prisma = new PrismaClient()

const server = new ApolloServer({
  typeDefs: fs.readFileSync(
    path.join(__dirname, 'schema.graphql'),
    'utf8'
  ),
  resolvers,
  context: {
    prisma,
  }
})
```

Awesome! Now, the `context` object that's passed into all your GraphQL resolvers is being initialized right here and because you're attaching an instance of `PrismaClient` (as `prisma`) to it when the `GraphQLServer` is instantiated, you'll now be able to access `context.prisma` in all of your resolvers.

Finally, it's time to refactor your resolvers. Again, we encourage you to type these changes yourself so that you can get used to Prisma's autocompletion and how to leverage that to intuitively figure out what resolvers should be on your own.

- Open `index.js` and remove the `links` array entirely, as well as the `idCount` variable – you don't need those any more since the data will now be stored in an actual database.

Next, you need to update the implementation of the resolver functions because they're still accessing the variables that were just deleted. Plus, you now want to return actual data from the database instead of local dummy data.

- Still in `index.js`, update the `resolvers` object to look as follows:

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

```
const resolvers = {
  Query: {
    info: () => `This is the API of a Hackernews Clone`,
    feed: async (parent, args, context) => {
      return context.prisma.link.findMany()
    },
  },
  Mutation: {
    post: (parent, args, context, info) => {
      const newLink = context.prisma.link.create({
        data: {
          url: args.url,
          description: args.description,
        },
      })
      return newLink
    },
  },
}
```

Now let's understand how these new resolvers are working!

Understanding the `feed` resolver

The `feed` resolver is implemented as follows:

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

```
feed: async (parent, args, context, info) => {
```

```
  return context.prisma.link.findMany()
},
```

It accesses the `prisma` object via the `context` argument we discussed a moment ago. As a reminder, this is actually an entire `PrismaClient` instance that's imported from the generated `@prisma/client` library, effectively allowing you to access your database through the Prisma Client API you set up in chapter 4.

Now, you should be able to imagine the complete system and workflow of a Prisma/GraphQL project, where our Prisma Client API exposes a number of database queries that let you read and write data in the database.

Understanding the `post` resolver

The `post` resolver now looks like this:

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

```
post: (parent, args, context) => {
  const newLink = context.prisma.link.create({
    data: {
      url: args.url,
      description: args.description,
    },
  })
  return newLink
},
```

Similar to the `feed` resolver, you're simply invoking a function on the `PrismaClient` instance which is attached to the `context`.

You're calling the `create` method on a `link` from your Prisma Client API. As arguments, you're passing the data that the resolvers receive via the `args` parameter.

So, to summarize, Prisma Client exposes a CRUD API for the models in your datamodel for you to read and write in your database. These methods are auto-generated based on your model definitions in `schema.prisma`.

Testing the new implementation

With these code changes, you can now go ahead and test if the new implementation with a database works as expected. As usual, run the following command in your terminal to start the GraphQL server:

$ .../hackernews-node

```
node src/index.js
```

Then, open the GraphQL Playground at `http://localhost:4000`. You can send the same `feed` query and `post` mutation as before. However, the difference is that this time the submitted links will be persisted in your SQLite database. Therefore, if you restart the server, the `feed` query will keep returning the same links.

Exploring your data in Prisma Studio

Prisma ships with a powerful database GUI where you can interact with your data: Prisma Studio.

Prisma Studio is different from a typical database GUI (such as TablePlus) in that it provides a layer of abstraction which allows you to see your data represented as it is in your Prisma data model.

This is one of the several ways that Prisma bridges the gap between how you structure and interact with your data in your application and how it is actually structured and represented in the underlying database. One major benefit of this is that it helps you to build intuition and understanding of these two linked but separate layers over time.

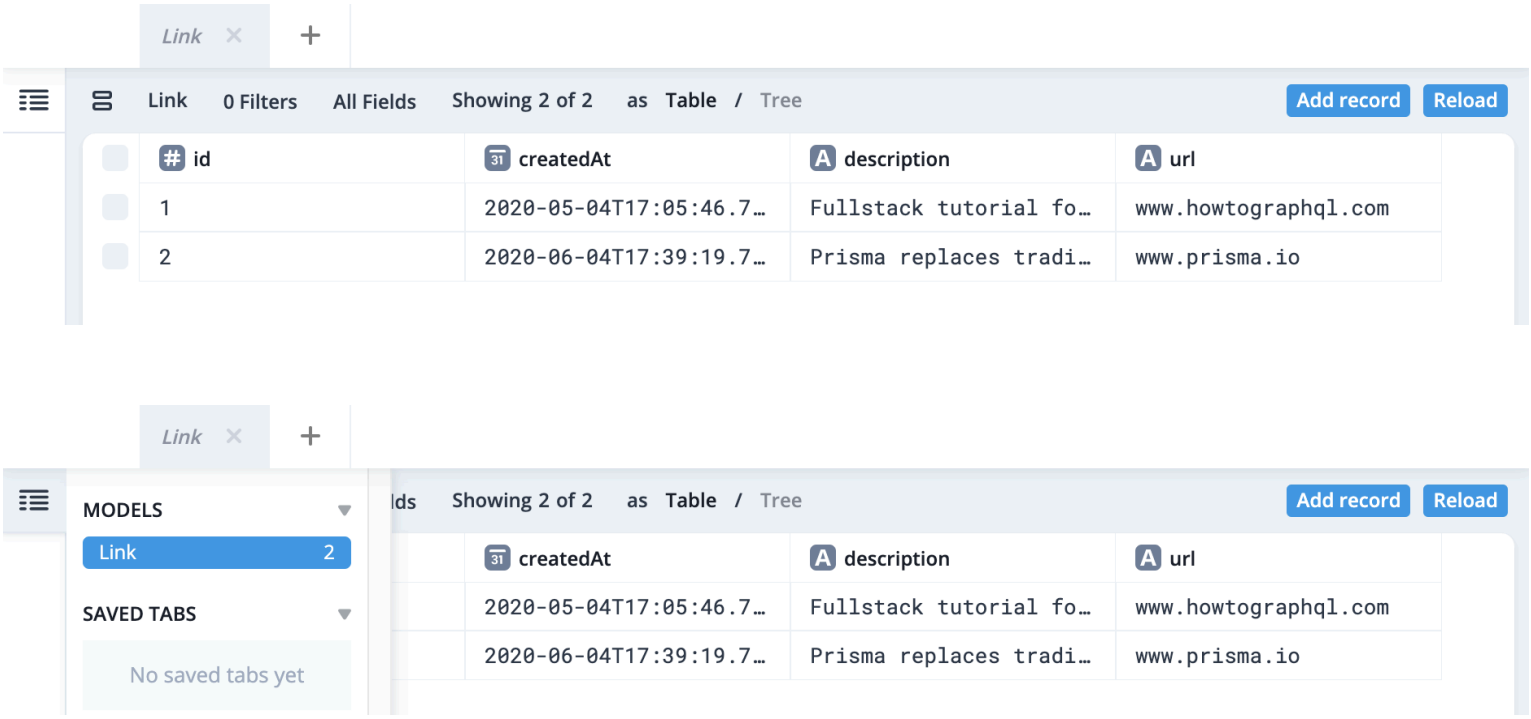
Let's run Prisma Studio and see it in action!

- Run the following command in your terminal

.../hackernews-node

```
npx prisma studio
```

Running the command should open a tab in your browser automatically (running on `http://localhost:5555`) where you will see the following interface. Notice that you see a tab for your `Link` model and can also explore all models by hovering on the far left menu:



UNLOCK THE NEXT CHAPTER

What is the purpose of the context argument in GraphQL resolvers?

- ○ It always provides access to a database
- ○ It carries the query arguments
- ○ It is used for authentication
- ✓ It lets resolvers communicate with each other

이것도 생각 맞았네!

Skip

Edit on Github

Authentication

What? 이게 뭐였지 ..?

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.

Prisma schema: DB의 어떻게 표시할 것인가 (Prisma Studio)

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:

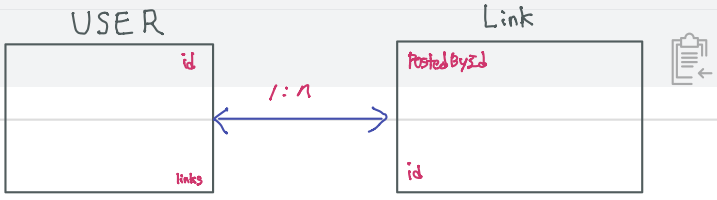
.../hackernews-node/prisma/data model.prisma

```
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[]
}
```

fk

User에 들어있는 PK

foreign key



- getAllLinksByUserId(user.id): Link[]
- getOwner(link.id): User

Now we start to see even more how Prisma helps you to reason about your data in a way that is more aligned with how it is represented in the underlying database.

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 dev --name "add-user-model" --preview-feature
```

migrate message

This command has now generated your second migration inside of `prisma/migrations`, and you can start to see how this becomes a historical record of how your database evolves over time.

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
```

re-generate prisma-client

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 as follows:

.../hackernews-node/src/schema.graphql

```
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

```
type Link {
  id: ID!
```

```
  description: String!
  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.findUnique({ where: { email: args.em
  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,
  }
}

module.exports = {
  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 getTokenPayload(token) {
  return jwt.verify(token, APP_SECRET);
}

function getUserId(req, authToken) {
  if (req) {
    const authHeader = req.headers.authorization;
    if (authHeader) {
      const token = authHeader.replace('Bearer ', '');
      if (!token) {
        throw new Error('No token found');
      }
```

```
      const { userId } = getTokenPayload(token);
      return userId;
    }
  } else if (authToken) {
    const { userId } = getTokenPayload(authToken);
    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 function will throw an *exception*. You can therefore use it to "protect" the resolvers which require authentication.

- 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 { getUserId } = require('./utils');

const server = new ApolloServer({
  typeDefs: fs.readFileSync(
    path.join(__dirname, 'schema.graphql'),
    'utf8'
  ),
  resolvers,
  context: ({ req }) => {
    return {
      ...req,
      prisma,
      userId:
        req && req.headers.authorization
          ? getUserId(req)
          : null
    };
  }
});
```

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.

Requiring authentication for the `post` mutation

Before you're going to test your authentication flow, make sure to complete your 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

```
async function post(parent, args, context, info) {
  const { userId } = context;

  return await 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 tutorial:

```
Link: {
  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.findUnique({ where: { id: parent.id } }).postedB
}

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.findUnique({ where: { id: parent.id } }).links()
}

module.exports = {
  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

The very first thing you'll do is test the `signup` mutation and thereby create a new

`User` in the database.

- 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:

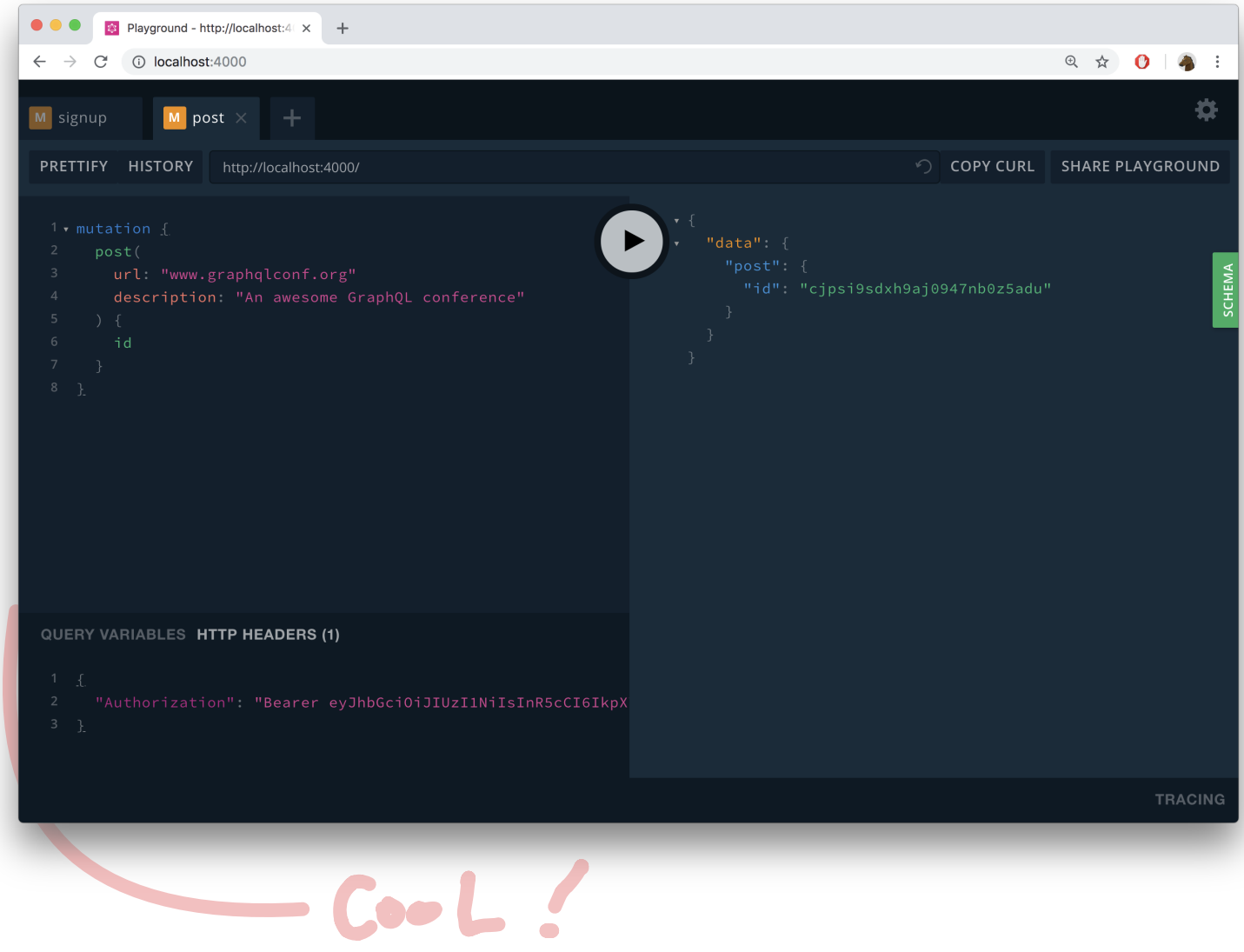
```
{
  "Authorization": "Bearer __TOKEN__"
}
```

Whenever you're now sending a query/mutation from that tab, it will carry the

authentication token.

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

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



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.

To verify everything worked, you can send the following `login` mutation:

```
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"
          }
        ]
      }
    }
  }
}
```

Which HTTP header field carries the authentication token?

- ○ Cache-Control
- ○ Token
- ✓ ○ Authorization
- ○ Authentication

Skip

Edit on Github

Realtime GraphQL Subscriptions

In this section, you'll learn how you can bring realtime functionality into your app by implementing GraphQL subscriptions. The goal is to implement two subscriptions to be exposed by your GraphQL server:

- Send realtime updates to subscribed clients when a new `Link` element is *created*
- Send realtime updates to subscribed clients when an existing `Link` element is *upvoted*

What are GraphQL subscriptions?

Subscriptions are a GraphQL feature that allows a server to send data to its clients when a specific *event* happens. Subscriptions are usually implemented with WebSockets. In that setup, the server maintains a steady connection to its subscribed client. This also breaks the "Request-Response-Cycle" that were used for all previous interactions with the API.

Instead, the client initially opens up a long-lived connection to the server by sending a *subscription query* that specifies which *event* it is interested in. Every time this particular event happens, the server uses the connection to push the event data to the subscribed client(s).

Implementing GraphQL subscriptions

We will be using `PubSub` from the `apollo-server` library that we have already been using for our GraphQL server to implement subscriptions to the following *events*:

- A new model is **created**
- An existing model **updated**
- An existing model is **deleted**

You will do this by first adding an instance of `PubSub` to the context, just as we did with `PrismaClient`, and then calling its methods in the resolvers that handle each of the above events.

Setting up `PubSub`

- Open your `index.js` file where we instantiate the server and add the following code:

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

```
// ... previous import statements
const { PubSub } = require('apollo-server')

const pubsub = new PubSub()
```

Here, you're creating an instance of `PubSub` and storing it in the variable `pubsub`, just as we stored an instance of `PrismaClient` in the variable `prisma`.

- Now, in the same file, add `pubsub` to the context, just as did with `prisma`:

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

```
const server = new ApolloServer({
  typeDefs: fs.readFileSync(
    path.join(__dirname, 'schema.graphql'),
    'utf8'
  ),
  resolvers,
  context: ({ req }) => {
    return {
      ...req,
      prisma,
      pubsub
      userId:
        req && req.headers.authorization
          ? getUserId(req)
          : null
```

```
    };
  }
});
```

Great! Now we can access the methods we need to implement our subscriptions from inside our resolvers via `context.pubsub`!

Subscribing to new `Link` elements

Alright – let's go ahead and implement the subscription that allows your clients to subscribe to newly created `Link` elements.

Just like with queries and mutations, the first step to implement a subscription is to extend your GraphQL schema definition.

- Open your application schema and add the `Subscription` type:

.../hackernews-node/src/schema.graphql

```
type Subscription {
  newLink: Link
}
```

Next, go ahead and implement the resolver for the `newLink` field. Resolvers for subscriptions are slightly different than the ones for queries and mutations:

1. Rather than returning any data directly, they return an `AsyncIterator` which subsequently is used by the GraphQL server to push the event data to the client.
2. Subscription resolvers are wrapped inside an object and need to be provided as the value for a `subscribe` field. You also need to provide another field called `resolve` that actually returns the data from the data emitted by the `AsyncIterator`.

To adhere to the modular structure of your resolver implementation, first create a new file called `Subscription.js`:

$ .../hackernews-node

```
touch src/resolvers/Subscription.js
```

Here's how you need to implement the subscription resolver in that new file:

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

```
function newLinkSubscribe(parent, args, context, info) {
  return context.pubsub.asyncIterator("NEW_LINK")
}

const newLink = {
  subscribe: newLinkSubscribe,
  resolve: payload => {
    return payload
  },
}

module.exports = {
  newLink,
}
```

The code seems pretty straightforward. As mentioned before, the subscription resolver is provided as the value for a `subscribe` field inside a plain JavaScript object.

Now you can see how we access `pubsub` on the `context` and call `asyncIterator()` passing the string `"NEW_LINK"` into it. This function is used to resolve subscriptions and push the event data.

Adding subscriptions to your resolvers

The last thing we need to do for our subscription implementation itself is to actually

call this function inside of a resolver!

- Pop over to `Mutation.js` and locate your `post` resolver function, adding the following call to `pubsub.publish()` right before we return our `newLink`:

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

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

  const newLink = await context.prisma.link.create({
    data: {
      url: args.url,
      description: args.description,
      postedBy: { connect: { id: userId } },
    }
  })
  context.pubsub.publish("NEW_LINK", newLink)

  return newLink
}
```

Now you can see how we pass the same string to the `publish` method as you added in your `newLinkSubscribe` function just above, along with passing in the `newLink` as a second argument!

Ok, I'm sure you're dying to test out your brand-spanking new Subscription! All we need to do now is make sure your GraphQL server knows about your changes.

- Open `index.js` and add an import statement for the `Subscription` module to the top of the file:

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

```
const Subscription = require('./resolvers/Subscription')
```

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

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

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

Testing subscriptions

With all the code in place, it's time to test your realtime API ⚡ You can do so by using two instances (i.e. tabs or windows) of the GraphQL Playground at once.

- If you haven't already, restart the server by first killing it with **CTRL+C**, then run `node src/index.js` again.
- Next, open two browser windows and navigate both to the endpoint of your GraphQL server: **`http://localhost:4000`**.

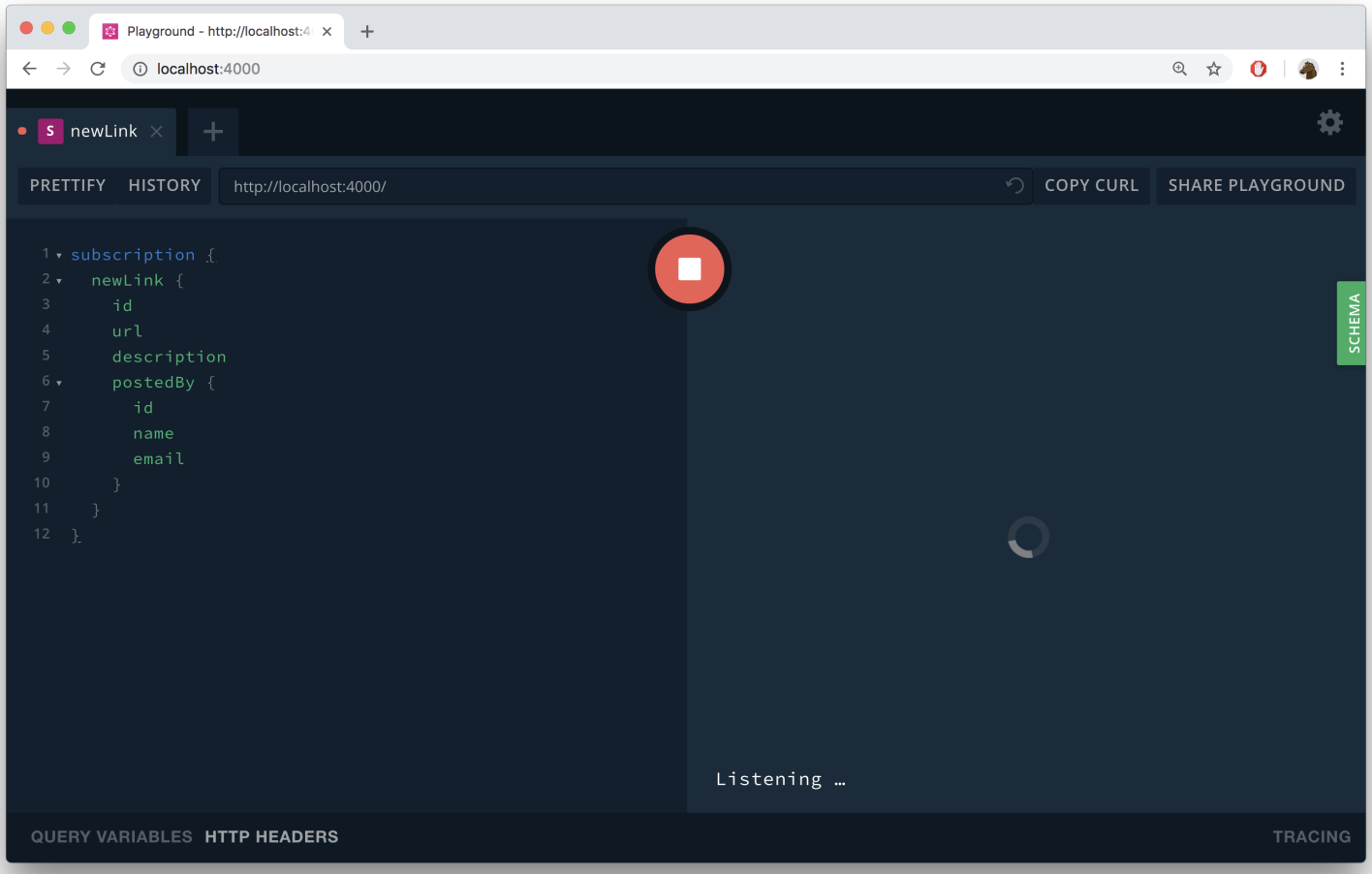
As you might guess, you'll use one Playground to send a subscription query and thereby create a permanent websocket connection to the server. The second one will be used to send a `post` mutation which triggers the subscription.

- In one Playground, send the following subscription:

```
subscription {
  newLink {
    id
    url
    description
    postedBy {
      id
      name
      email
    }
```

```
    }
}
```

In contrast to what happens when sending queries and mutations, you'll not immediately see the result of the operation. Instead, there's a loading spinner indicating that it's waiting for an event to happen.

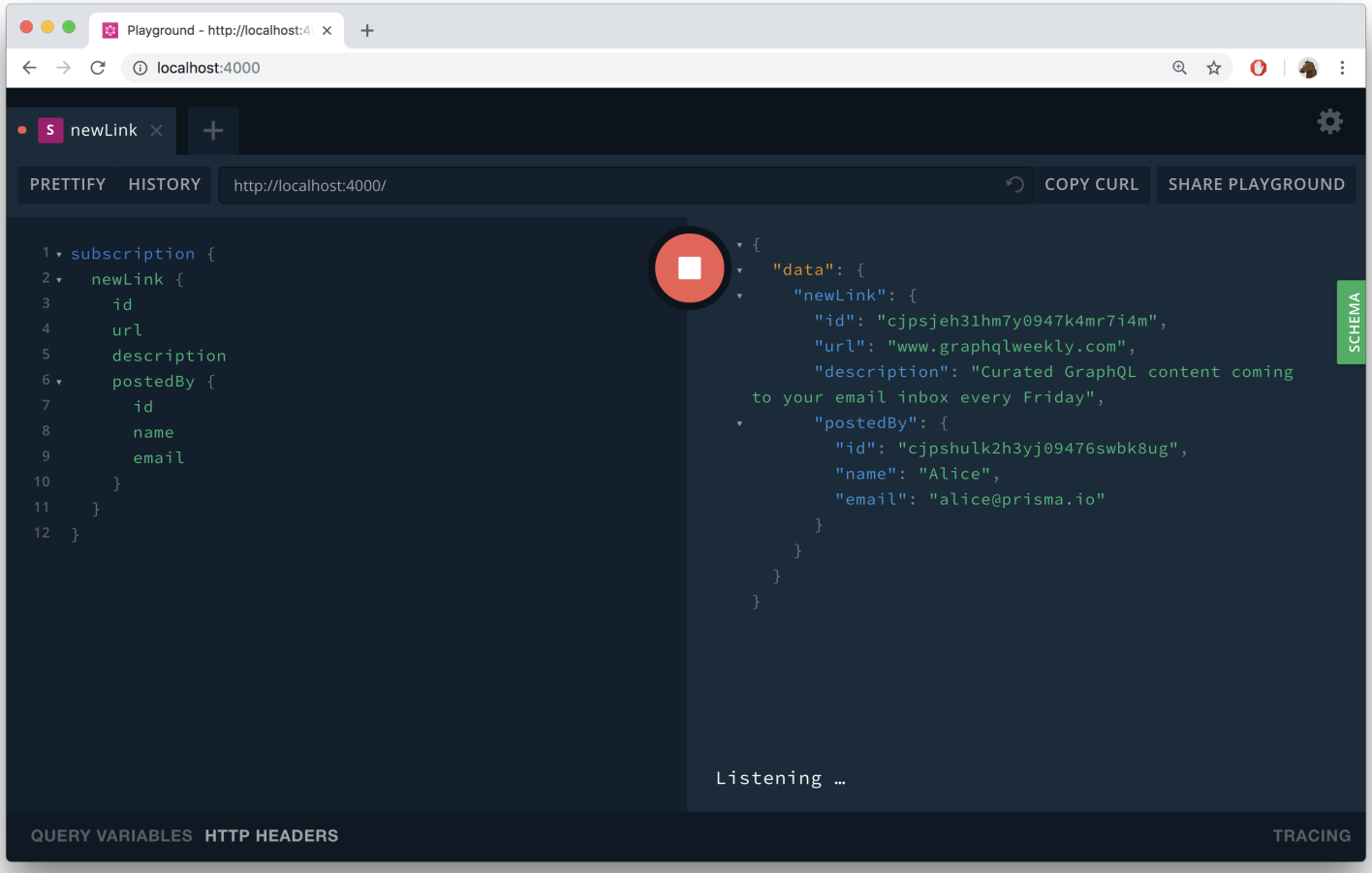


Time to trigger a subscription event.

- Send the following `post` mutation inside a GraphQL Playground. Remember that you need to be authenticated for that (see the previous chapter to learn how that works):

```
mutation {
  post(url: "www.graphqlweekly.com", description: "Curated GraphQL content co
    id
  }
}
```

Now observe the Playground where the subscription was running:



Adding a voting feature

Implementing a `vote` mutation

The next feature to be added is a voting feature which lets users *upvote* certain links. The very first step here is to extend your Prisma data model to represent votes in the

database.

- Open `prisma/schema.prisma` and adjust it to look as follows:

.../hackernews-node/prisma/schema.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?
  votes       Vote[]
}

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

model Vote {
  id     Int  @id @default(autoincrement())
  link   Link @relation(fields: [linkId], references: [id])
  linkId Int
  user   User @relation(fields: [userId], references: [id])
  userId Int

  @@unique([linkId, userId])
}
```

As you can see, you added a new `Vote` type to the data model. It has one-to-many relationships to the `User` and the `Link` type.

- Now migrate your database schema with the following commands:

```
npx prisma migrate dev --name 'add-vote-model' --preview-feature
```

To apply the changes and update your Prisma Client API so it exposes CRUD queries for the new `Vote` model, regenerate `PrismaClient`.

- Run the following command in your terminal:

$ .../hackernews-node

```
npx prisma generate
```

Now, with the process of schema-driven development in mind, go ahead and extend the schema definition of your application schema so that your GraphQL server also exposes a `vote` mutation:

.../hackernews-node/src/schema.graphql

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

- The referenced `Vote` type also needs to be defined in the GraphQL schema:

.../hackernews-node/src/schema.graphql

```
type Vote {
  id: ID!
  link: Link!
  user: User!
}
```

It should also be possible to query all the `votes` from a `Link`, so you need to adjust the `Link` type in `schema.graphql` as well.

Open `schema.graphql` and add the `votes` field to `Link`:

.../hackernews-node/src/schema.graphql

```
type Link {
  id: ID!
  description: String!
  url: String!
  postedBy: User
  votes: [Vote!]!
}
```

You know what's next: Implementing the corresponding resolver functions.

Add the following function to `src/resolvers/Mutation.js`:

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

```
async function vote(parent, args, context, info) {
  // 1
  const userId = getUserId(context)

  // 2
  const vote = await context.prisma.vote.findUnique({
    where: {
      linkId_userId: {
        linkId: Number(args.linkId),
        userId: userId
      }
    }
  })

  if (Boolean(vote)) {
    throw new Error(`Already voted for link: ${args.linkId}`)
  }
```

```
  // 3
  const newVote = context.prisma.vote.create({
    data: {
      user: { connect: { id: userId } },
      link: { connect: { id: Number(args.linkId) } },
    }
  })
  context.pubsub.publish("NEW_VOTE", newVote)

  return newVote
}
```

Here is what's going on:

1. Similar to what you're doing in the `post` resolver, the first step is to validate the incoming JWT with the `getUserId` helper function. If it's valid, the function will return the `userId` of the `User` who is making the request. If the JWT is not valid, the function will throw an exception.
2. To protect against those pesky "double voters" (or honest folks who accidentally click twice), you need to check if the vote already exists or not. First, you try to fetch a vote with the same `linkId` and `userId`. If the vote exists, it will be stored in the `vote` variable, resulting in the boolean `true` from your call to `Boolean(vote)` — throwing an error kindly telling the user that they already voted.
3. If that `Boolean(vote)` call returns `false`, the `vote.create` method will be used to create a new `Vote` that's *connected* to the `User` and the `Link`.

Also, don't forget to adjust the export statement to include the `vote` resolver in the module:

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

```
module.exports = {
  post,
  signup,
  login,
  vote,
```

```
}
```

You also need to account for the new relations in your GraphQL schema:

- `votes` on `Link`
- `user` on `Vote`
- `link` on `Vote`

Similar to before, you need to implement resolvers for these.

Open `Link.js` and add the following function to it:

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

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

Don't forget to include the new resolver in the exports:

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

```
module.exports = {
  postedBy,
  votes,
}
```

Finally you need to resolve the relations from the `Vote` type.

- Create a new file called `Vote.js` inside `resolvers`:

$ .../hackernews-node

```
touch src/resolvers/Vote.js
```

Great job!

- Now add the following code to it:

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

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

function user(parent, args, context) {
  return context.prisma.vote.findUnique({ where: { id: parent.id } }).user()
}

module.exports = {
  link,
  user,
}
```

Finally the `Vote` resolver needs to be included in the main `resolvers` object in `index.js`.

- Open `index.js` and add a new import statement to its top:

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

```
const Vote = require('./resolvers/Vote')
```

- Finally, include the `Vote` resolver in the `resolvers` object:

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

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

Your GraphQL API now accepts `vote` mutations! 👏

Subscribing to new votes

The last task in this chapter is to add a subscription that fires when new `Vote`s are being created. You'll use an analogous approach as for the `newLink` query for that.

- Add a new field to the `Subscription` type of your GraphQL schema:

.../hackernews-node/src/schema.graphql

```
type Subscription {
  newLink: Link
  newVote: Vote
}
```

Next, you need to add the subscription resolver function.

- Add the following code to `Subscription.js`:

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

```
function newVoteSubscribe(parent, args, context, info) {
  return context.pubsub.asyncIterator("NEW_VOTE")
}

const newVote = {
  subscribe: newVoteSubscribe,
  resolve: payload => {
    return payload
  },
}
```

- And update the export statement of the file accordingly:

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

```
module.exports = {
  newLink,
  newVote,
}
```

All right, that's it! You can now test the implementation of your `newVote` subscription!

- 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, open a new Playground with the GraphQL CLI by running `graphql playground`.

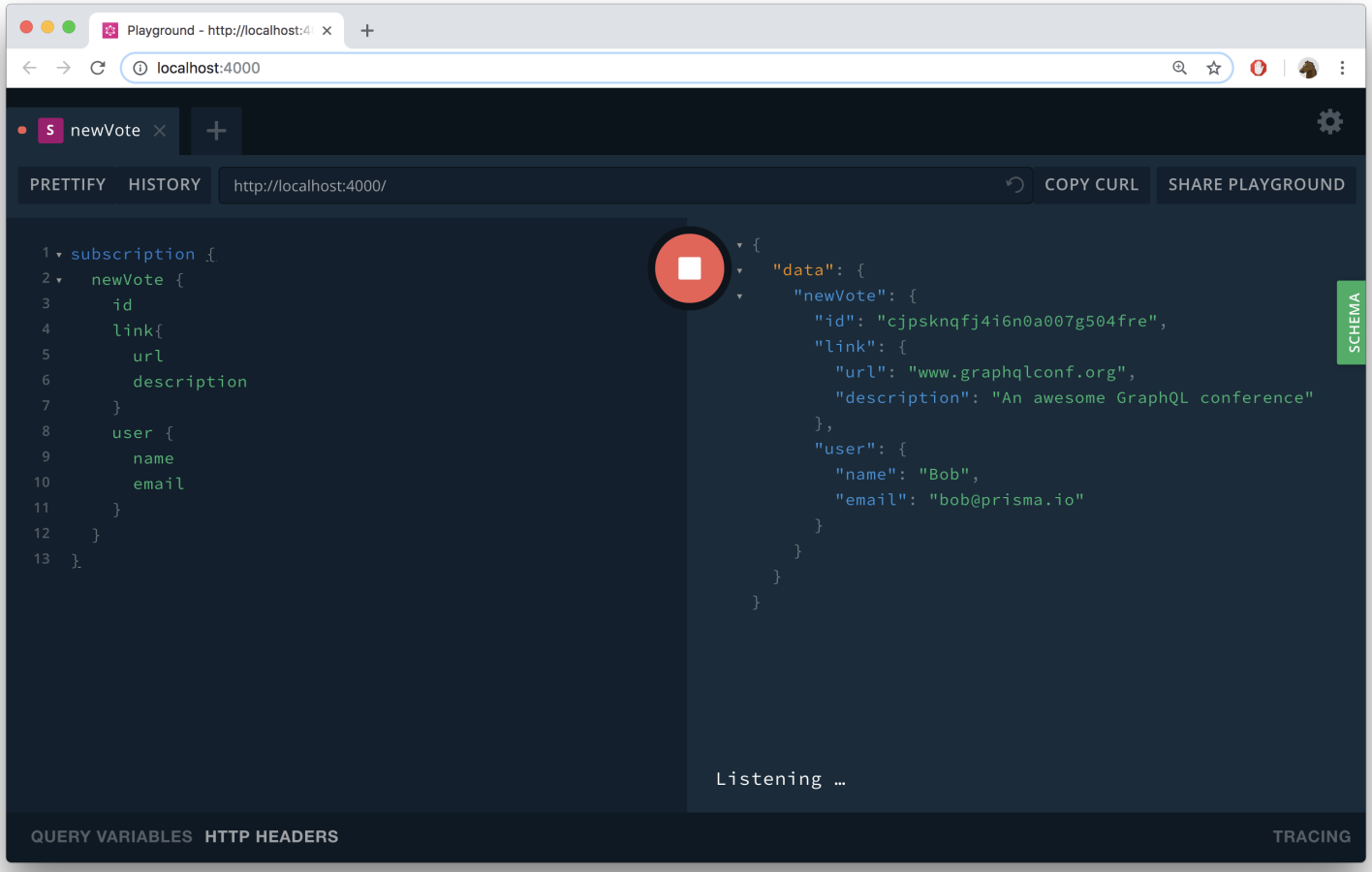
You can use the following subscription for that:

```
subscription {
  newVote {
    id
    link {
      url
      description
    }
    user {
```

```
      name
      email
    }
  }
}
```

If you're unsure about writing one yourself, here's a sample `vote` mutation you can use. You'll need to replace the `__LINK_ID__` placeholder with the `id` of an actual `Link` from your database. Also, make sure that you're authenticated when sending the mutation.

```
mutation {
  vote(linkId: "__LINK_ID__") {
    link {
      url
      description
    }
    user {
      name
      email
    }
  }
}
```



UNLOCK THE NEXT CHAPTER

Which of the following statements is true?

- ◯ Subscriptions follow a request-response-cycle
- ◯ Subscriptions are best implemented with MailChimp

- [x] Subscriptions are typically implemented via WebSockets
- [ ] Subscriptions are defined on the 'Query' type and annotated with the @realtime-directive

Skip

Edit on Github

Filtering, Pagination & Sorting

This is an exciting section of the tutorial where you'll implement some key features of many robust APIs! The goal is to allow clients to constrain the list of `Link` elements returned by the `feed` query by providing filtering and pagination parameters.

Let's jump in! 🚀

Filtering

By using `PrismaClient`, you'll be able to implement filtering capabilities to your API without too much effort. Similarly to the previous chapters, the heavy-lifting of query resolution will be performed by Prisma. All you need to do is forward incoming queries to it.

The first step is to think about the filters you want to expose through your API. In your case, the `feed` query in your API will accept a *filter string*. The query then should only return the `Link` elements where the `url` *or* the `description` *contain* that filter string.

- Go ahead and add the `filter` string to the `feed` query in your application schema:

.../hackernews-node/src/schema.graphql

```
type Query {
  info: String!
  feed(filter: String): [Link!]!
}
```

Next, you need to update the implementation of the `feed` resolver to account for the new parameter clients can provide.

- Open `src/resolvers/Query.js` and update the `feed` resolver to look as follows:

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

```
async function feed(parent, args, context, info) {
  const where = args.filter
    ? {
      OR: [
        { description: { contains: args.filter } },
        { url: { contains: args.filter } },
      ],
    }
    : {}

  const links = await context.prisma.link.findMany({
    where,
  })

  return links
}
```

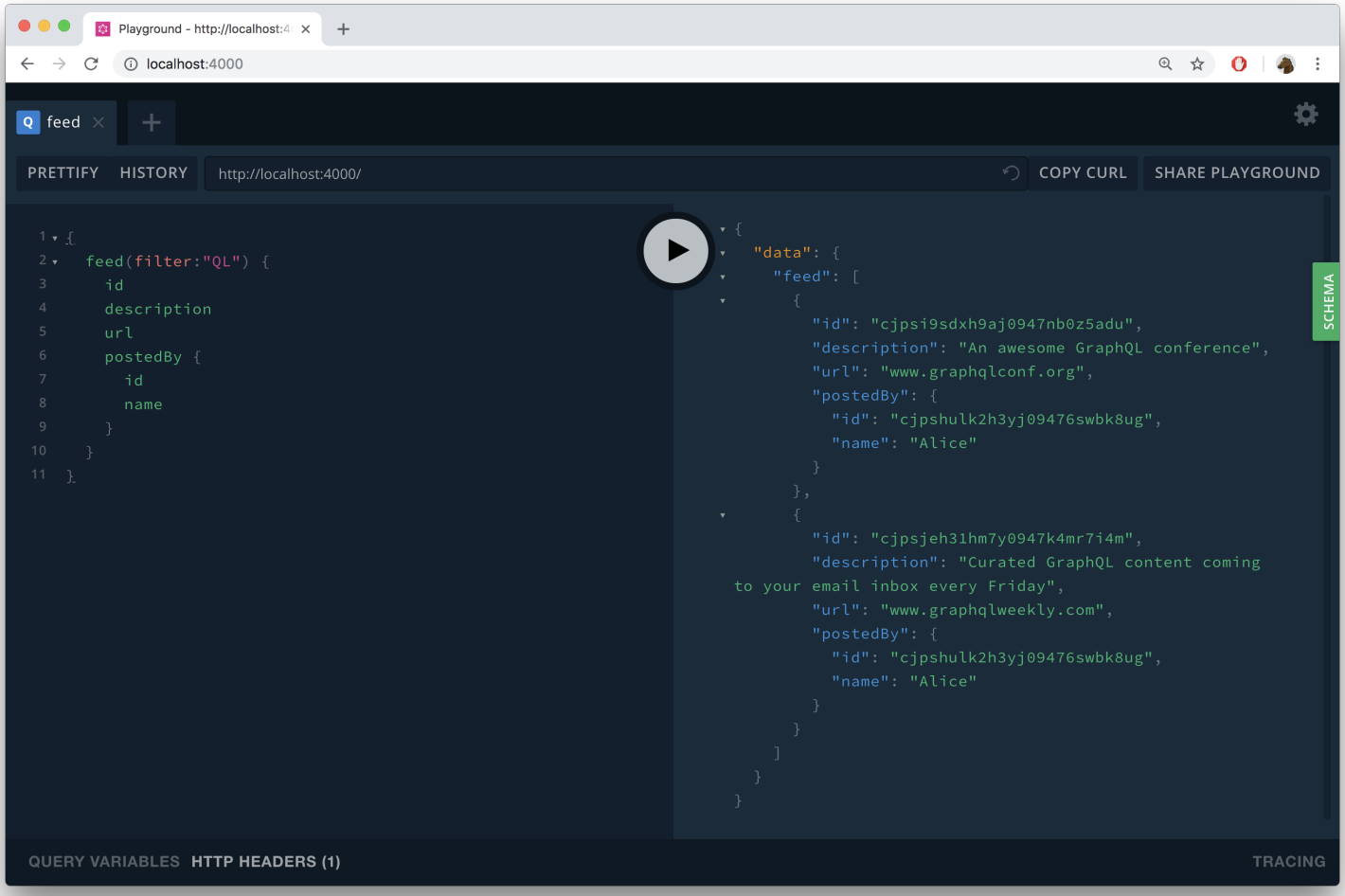
If no `filter` string is provided, then the `where` object will be just an empty object and no filtering conditions will be applied by Prisma Client when it returns the response for the `links` query.

In cases where there is a `filter` carried by the incoming `args`, you're constructing a `where` object that expresses our two filter conditions from above. This `where` argument is used by Prisma to filter out those `Link` elements that don't adhere to the specified conditions.

That's it for the filtering functionality! Go ahead and test your filter API - here's a sample query you can use:

```
query {
  feed(filter: "QL") {
    id
    description
```

```
    url
    postedBy {
      id
      name
    }
  }
}
```



Pagination

Pagination is a tricky topic in API design. On a high-level, there are two major approaches for tackling it:

- **Limit-Offset**: Request a specific *chunk* of the list by providing the indices of the items to be retrieved (in fact, you're mostly providing the start index (*offset*) as well as a count of items to be retrieved (*limit*)).
- **Cursor-based**: This pagination model is a bit more advanced. Every element in the

list is associated with a unique ID (the *cursor*). Clients paginating through the list then provide the cursor of the starting element as well as a count of items to be retrieved.

Prisma supports both pagination approaches (read more in the docs). In this tutorial, you're going to implement limit-offset pagination.

> **Note**: You can read more about the ideas behind both pagination approaches here.

Limit and offset have different names in the Prisma API:

- The *limit* is called `take`, meaning you're "taking" `x` elements after a provided start index.
- The *start index* is called `skip`, since you're skipping that many elements in the list before collecting the items to be returned. If `skip` is not provided, it's `0` by default. The pagination then always starts from the beginning of the list.

So, go ahead and add the `skip` and `take` arguments to the `feed` query.

- Open your application schema and adjust the `feed` query to accept `skip` and `take` arguments:

.../hackernews-node/src/schema.graphql

```
type Query {
  info: String!
  feed(filter: String, skip: Int, take: Int): [Link!]!
}
```

Now, on to the resolver implementation.

- In `src/resolvers/Query.js`, adjust the implementation of the `feed` resolver:

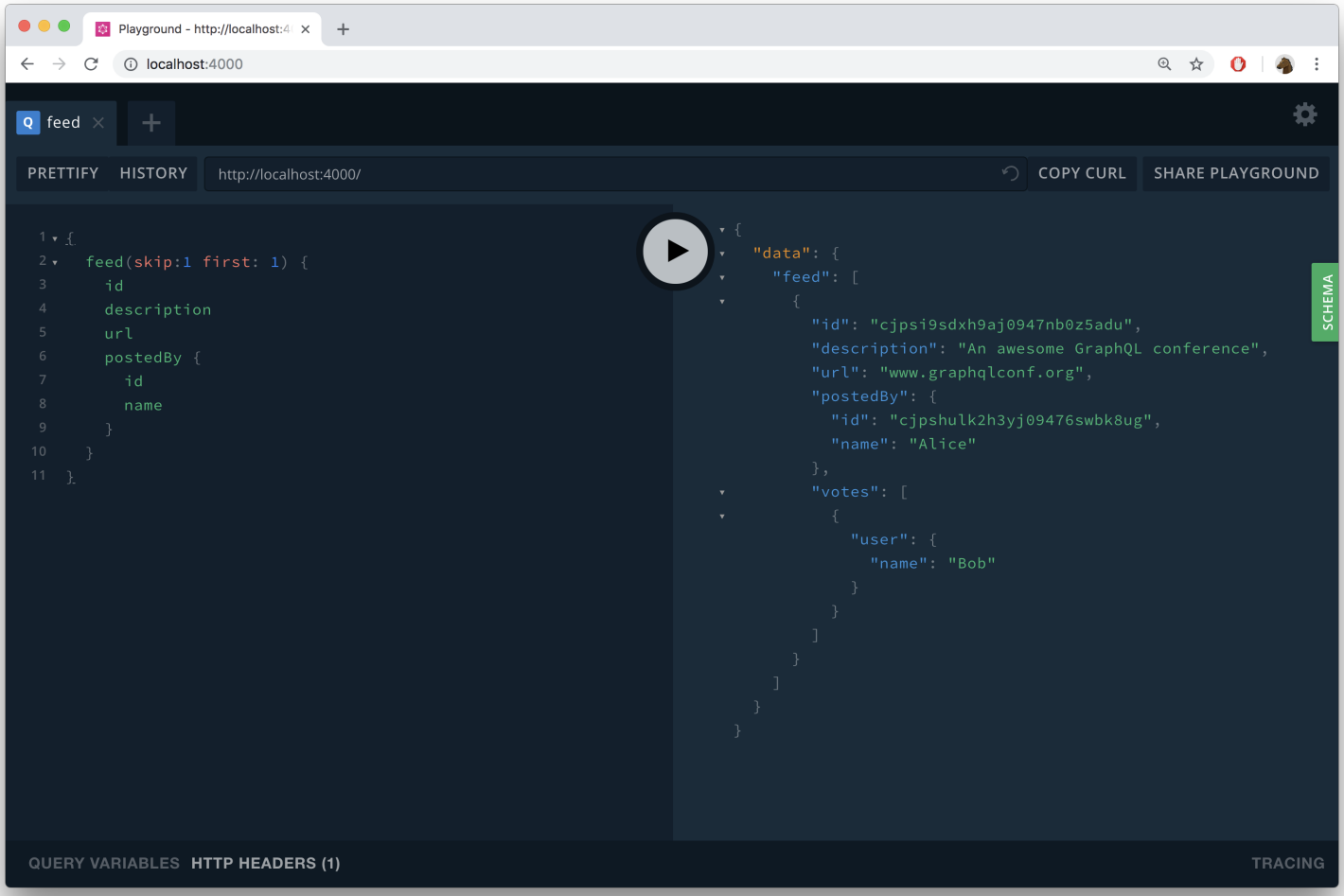
.../hackernews-node/src/resolvers/Query.js

```
async function feed(parent, args, context, info) {
  const where = args.filter
    ? {
      OR: [
        { description: { contains: args.filter } },
        { url: { contains: args.filter } },
      ],
    }
    : {}

  const links = await context.prisma.link.findMany({
    where,
    skip: args.skip,
    take: args.take,
  })

  return links
}
```

Really all that's changing here is that the invocation of the `links` query now receives two additional arguments which might be carried by the incoming `args` object. Again, Prisma will take care of the rest.

You can test the pagination API with the following query which returns the second `Link` from the list:

```
query {
  feed(take: 1, skip: 1) {
    id
    description
    url
  }
}
```



Sorting

With Prisma, it is possible to return lists of elements that are sorted (*ordered*) according to specific criteria. For example, you can order the list of `Link`s alphabetically by their `url` or `description`. For the Hacker News API, you'll leave it up to the client to decide how exactly it should be sorted and thus include all the ordering options from the Prisma API in the API of your GraphQL server. You can do so by creating an `input` type and an enum to represent the ordering options.

- Add the following enum definition to `schema.graphql`:

.../hackernews-node/src/schema.graphql

```
input LinkOrderByInput {
  description: Sort
  url: Sort
  createdAt: Sort
```

```
}

enum Sort {
  asc
  desc
}
```

This represents the various ways that the list of `Link` elements can be sorted.

- Now, adjust the `feed` query again to include the `orderBy` argument:

.../hackernews-node/src/schema.graphql

```
type Query {
  info: String!
  feed(filter: String, skip: Int, take: Int, orderBy: LinkOrderByInput): [Lir
}
```

The implementation of the resolver is similar to what you just did with the pagination API.

- Update the implementation of the `feed` resolver in `src/resolvers/Query.js` and pass the `orderBy` argument along to Prisma:

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

```
async function feed(parent, args, context, info) {
  const where = args.filter
    ? {
      OR: [
        { description: { contains: args.filter } },
        { url: { contains: args.filter } },
      ],
    }
    : {}
```

```
  const links = await context.prisma.link.findMany({
    where,
    skip: args.skip,
    take: args.take,
    orderBy: args.orderBy,
  })

  return links
}
```

Awesome! Here's a query that sorts the returned links by their creation dates:

```
query {
  feed(orderBy: { createdAt: asc }) {
    id
    description
    url
  }
}
```

Returning the total amount of `Link` elements

The last thing you're going to implement for your Hacker News API is the information *how many* `Link` elements are currently stored in the database. To do so, you're going to refactor the `feed` query a bit and create a new type to be returned by your API: `Feed`.

- Add the new `Feed` type to your GraphQL schema. Then also adjust the return type of the `feed` query accordingly:

.../hackernews-node/src/schema.graphql

```
type Query {
  info: String!
  feed(filter: String, skip: Int, take: Int, orderBy: LinkOrderByInput): Feed
}

type Feed {
  links: [Link!]!
```

```
  count: Int!
}
```

Now, go ahead and adjust the `feed` resolver again:

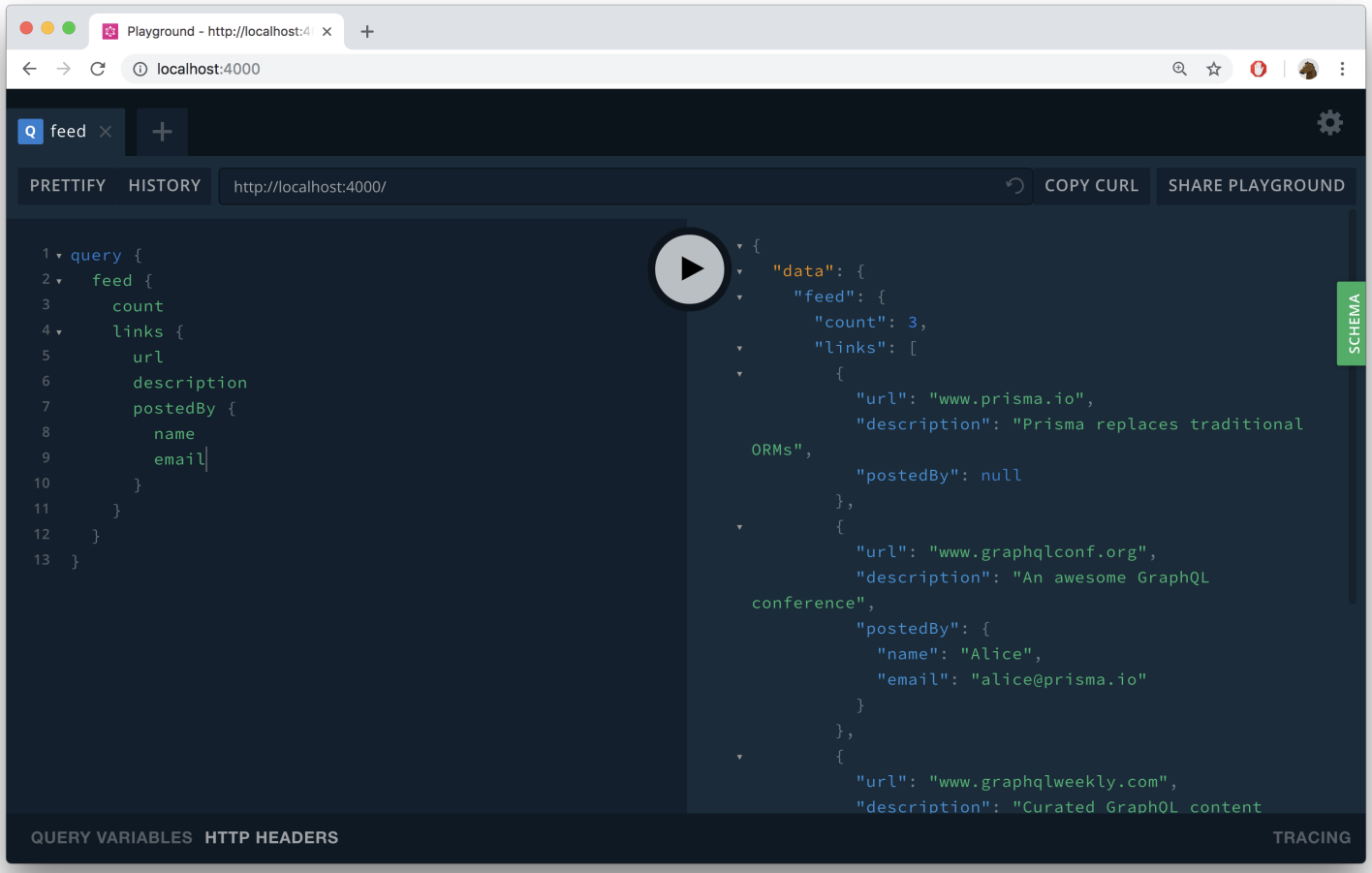
.../hackernews-node/src/resolvers/Query.js

```
async function feed(parent, args, context, info) {
  const where = args.filter
    ? {
      OR: [
        { description: { contains: args.filter } },
        { url: { contains: args.filter } },
      ],
    }
    : {}

  const links = await context.prisma.link.findMany({
    where,
    skip: args.skip,
    take: args.take,
    orderBy: args.orderBy,
  })

  const count = await context.prisma.link.count({ where })

  return {
    links,
    count,
  }
}
```

You can now test the revamped `feed` query as follows:

```
query {
  feed {
    count
    links {
      id
      description
```

```
      url
    }
  }
}
```



UNLOCK THE NEXT CHAPTER

Which arguments are typically used to paginate through a list in the Prisma Client API using limit-offset pagination?

- skip & take
- skip & orderBy
- take & where
- where & orderBy

Skip

Edit on Github

Summary

In this tutorial, you learned how to build a GraphQL server from scratch. The stack you used was based on Node.js, `apollo-server` and Prisma.

`apollo-server` is a fast and simple GraphQL server library built on top of Express.js. It comes with several features, such as out-of-the-box support for GraphQL Playgrounds and realtime GraphQL subscriptions.

The resolvers of your GraphQL server are implemented using Prisma Client which is responsible for database access.

If you want to dive deeper and become part of the awesome GraphQL community, here are a few resource and community recommendations for you:

- Prisma Blog: The blog regularly features new and interesting content about GraphQL, from community news to technical deep dives and various tutorials.
- GraphQL Weekly: A weekly GraphQL newsletter with news from the GraphQL ecosystem
- GraphQL Conf: The world's biggest gathering of GraphQL enthusiasts happening in the heart of Berlin
- Prisma Slack: A Slack team with vivid discussions around everything GraphQL & Prisma

Congratulations on completing the tutorial! We can't wait to see what you build.

Did you find this tutorial useful?

- ○ What tutorial?
- ○ Yes, I learned something!
- ○ No, I even forgot what I knew before!
- ○ Fish!

Skip

Edit on Github